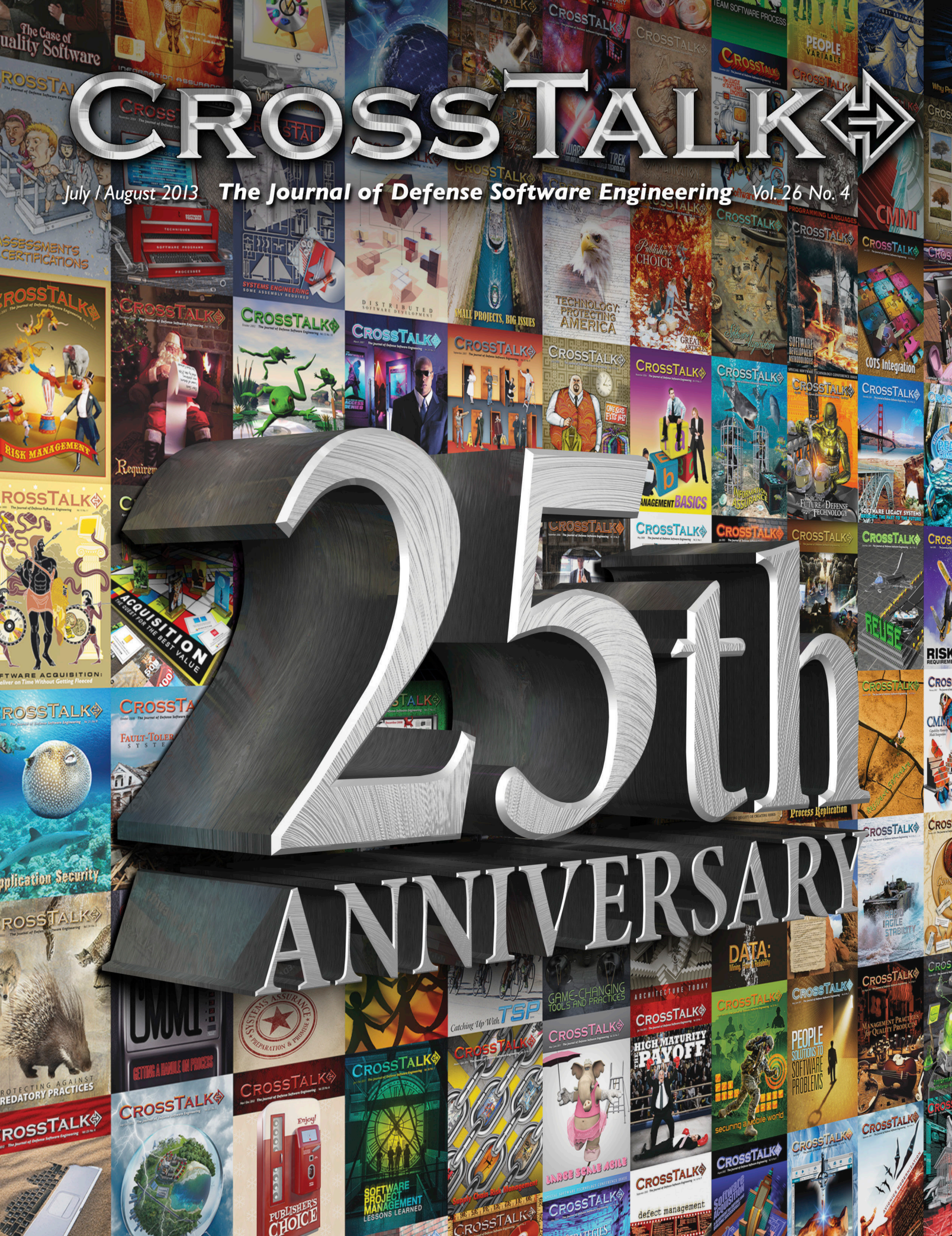

CROSSTALK
July / August 2013 The Journal of Defense Software Engineering Vol. 26 No. 4
25th
ANNIVERSARY

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
AUG 2013		**00-07-2013 to 00-08-2013**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk, The Journal of Defense Software Engineering. Volume 26, Number 4. July/August 2013	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **517 SMXS MXDEA,6022 Fir Ave Bldg 1238,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

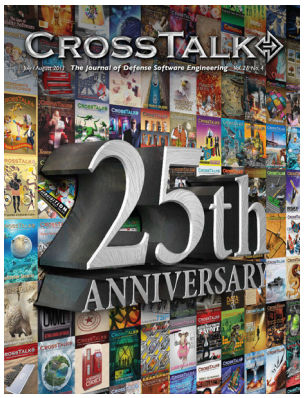
14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**40**	

TABLE OF CONTENTS

Departments



Cover Design by Kent Bingham

25th Anniversary Issue

CrossTalk

NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Publisher Justin T. Hill
Article Coordinator Lynne Wade
Managing Director David Erickson
Technical Program Lead Thayne M. Hill
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Art Director Kevin Kiernan

Phone 801-777-9828
E-mail Crosstalk.Articles@hill.af.mil
CrossTalk Online www.crosstalkonline.org

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Defense (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: National Cyber Security Division in the National Protection and Program Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of **CrossTalk** providing both editorial oversight and technical review of the journal. **CrossTalk's** mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

Subscriptions: Visit <www.crosstalkonline.org/subscribe> to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the **CrossTalk** editorial board prior to publication. Please follow the Author Guidelines, available at <www.crosstalkonline.org/submission-guidelines>. **CrossTalk** does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the authors and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with **CrossTalk.**

Trademarks and Endorsements: **CrossTalk** is an authorized publication for members of the DoD. Contents of **CrossTalk** are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services:
For questions or concerns about crosstalkonline.org web content or functionality contact the **CrossTalk** webmaster at 801-417-3000 or webmaster@luminpublishing.com.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

CrossTalk is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing <luminpublishing.com>. ISSN 2160-1577 (print); ISSN 2160-1593 (online)

CrossTalk would like to thank 309 SMXG for sponsoring this issue.

It is interesting and informative to reflect back over 25 years of software development. Twenty-five years ago the Software Crisis was raging and there were many engaged in trying to convert software code production from art to engineering science. The DoD was actively funding the pursuit of a solution. The term Software Crisis was first coined at a NATO Software Engineering Conference in 1968. It was the result of dramatic increases in computing power outpacing the ability of developers to produce working software. It is no wonder the DoD was interested in improving the odds of software being successful; at the time, approximately one in eight finished software projects were considered successful. It was this DoD effort to improve software development that originally funded the creation of CrossTalk as an information exchange forum.

Watts Humphrey published the CMM® 25 years ago in 1988 and as a book, "Managing the Software Process" the following year. This was the beginning of a lot of great work on software process improvement. CMM would later be followed by other great works by Watts Humphrey such as Team Software Process (TSP) and Personal Software Process (PSP). All along the way, CrossTalk has been there covering the transformation of the software industry from crisis to manageable and predictable software development. CrossTalk has published articles about many types of process improvement, some of which have been successful and others not so much. Many of us have witnessed firsthand this transformation of the software industry. We have seen the transformation from very limited process control to process control being the rule, not the exception. We have seen the progression from CMM to the CMMI®.

Today we continue to strive to improve quality and predictability while at the same time reducing cost. Unlike 25 years ago, we now have data and processes that support controlled predictable high-quality software development. We have all probably participated in the debates about what amount of process improvement/control is enough. As the Software Maintenance Group Director, I don't know the ultimate answer to the question; however we continue to pursue improved software predictability, quality and price.
This issue of CrossTalk is focused on just how things have changed over the last 25 years. I hope you enjoy the perspectives provided in this issue of CrossTalk.

Karl Rogers
Software Maintenance Group Director
309th Software Maintenance Group

Disclaimer:
CMMI® and CMM® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University

CrossTalk and Software—Past, Present and Future

A Twenty-Five Year Perspective

David A. Cook, Ph.D., Stephen F. Austin State University

Abstract. Since its initial issue, CrossTalk has helped guide software development throughout the DoD. As CrossTalk celebrates its 25th anniversary, it is educational to see how much software has changed and evolved over the lifetime of CrossTalk—and where the future might be leading us. This article discusses several of the forces that have shaped software and developmental languages over the last 25 years and also tries to see where the future will be taking us.

In the Beginning: Pre-CrossTalk

Although 25 years is a short span of time, it is actually a very long time in terms of software evolution. Twenty-Five years is over one-third the entire life span of computers—after all, the ENIAC only dates from 1946 [1].

One could also argue that some of the most important changes in computers and software occurred in the last 25 years—after all, the commercialization of the Internet did not begin until the mid 1990s. Standardization of TCP/IP itself did not begin until the 1980s [2]. The replacement of the large mainframe computers with desktop "microcomputers" did not happen until the late 1980s. Of course, lots of software development was accomplished prior to the existence of CrossTalk. In the early days of software development, however, it was normal for developers to need intimate knowledge of the target hardware.

Back in the 1950s and even into the 1960s, machine code was used for many applications—and the only tools available were assemblers. Even when working with assembly language (which was much simpler to understand than machine code), developers had to have extensive knowledge of the hardware that the final software would be deployed upon. The tools that were available during these early days were relatively simplistic. The developer was closely tied to not only a machine, but occasionally tied to a particular model and configuration. The interface between the developer and the hardware was direct—and hard to learn and master. The developer had to understand not just the problem space, but also had to be a master of the hardware. At best, an assembler abstracted away some of the hardware, but not all. Developers still were tied to hardware—and had to understand it to develop any code [3].

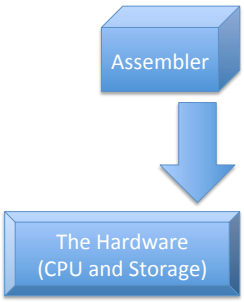


Figure 1 – Adding a tool to abstract away part of the actual computer

The Era of CrossTalk–The Early Days

The Quantity of Programming Languages

Twenty-five years ago, compilers and languages proliferated. There were many reasons for the creation of a new programming language [4], and the result was that by the 1980s, more than 2,000 programming languages existed [5]. Often companies or projects created a new language because their proposed software needed a combination of features not found in an existing language. Because the machines (and storage) of the time were limited, trying to add additional features to a language that already had features they might not need would simply increase compile time. Back in the 1980s it was not unusual for compile time to run to minutes per line! Adding new features to existing languages simply made compile time worse. It was more attractive to start fresh—and develop a language that had only the exact features needed for a project.

By the time CrossTalk came along, it was reasonably well recognized in the DoD that a minimal set of languages would make software maintenance easier but allow more transfer of knowledge and reuse of code throughout the DoD. Simply put, it is not cost efficient to maintain systems in thousands of languages, nor is it wise to have a software development workforce that is segmented by knowledge of so many niche languages.

While it was recognized that such a minimal set of languages needed to include some legacy languages (JOVIAL, CMS, Fortran, COBOL), the DoD also wanted to develop a language that it hoped would meet everybody's programming needs. During the infancy of CrossTalk, Ada was developed and heavily promoted by the DoD as a language that would unify software development needs. For numerous reasons (many political), Ada never became the huge success that the DoD envisioned. Commercial languages that dominate today's software development market include Java and C (and the descendants of C, such as C++ and C#). To understand the forces driving language design and language selection, it helps to examine a programming language from the perspective of what it provides to the developer.

The Quality of Programming Languages

Early high-level languages provided "machine transparency" to the developer. Without having to know and master such concepts as word size, memory size, how many registers were available, etc., the developer could spend less time concentrating on "what platform the solution will be implemented on" and more time on just understanding the problem. A "good" programming language let the programmer focus on the problem, rather than the hardware—but at the same time, provided enough features to permit the majority of general-purpose software tasks to be easily accomplished.

The earliest compilers were adequate for basic generalized programming needs. They provided the developer with a way to abstract themselves yet one step further away from the machine. In essence, the compilers were a tool that provided input to another tool (the assembler), which, in turn interfaced with the hardware.

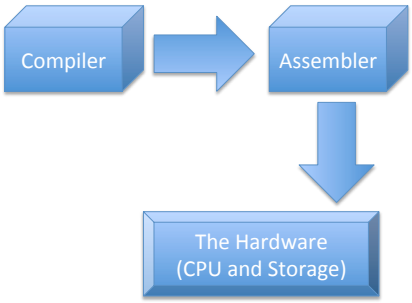


Figure 2 – Adding one more level between the developer and the hardware

One of the driving forces behind software development has been, oddly enough, a hardware force—Moore's Law [6]. Moore's Law (Gordon Moore was one of the co-founders of Intel) was that the number of components of an integrated circuit doubles about every two years. The law (more of an observation) has proven to be uncannily accurate over the last 50 years. And the law has been expanded to cover the capabilities of many digital electronic devices that are strongly linked to Moore's Law: processing speed, memory capacity, disk capacity, and even the number and size of pixels. Because this law says that everything doubles every two years, then the capacity of computers (in terms of speed, memory, and storage) is exponential. From Moore's Law comes what I refer to as Cook's Observation of Unwanted Space—every CPU cycle and byte of storage will eventually become used. Back in the 1960s, the Titan missile used less than 2,000 lines of code. The F-35 Joint Strike Fighter uses around 25 million [7].

The Recent Past and the Present

Software of Today

How is it possible that computer speed, memory, and storage are doubling every two years, but we are continuing to use mainstream languages (such as Java and C++) to develop modern software systems that demand more and more capabilities? We manage to accomplish this by continually updating just the language, but by continuing to create and update extensive libraries and templates to assist us with coding. Granted, we continue to update modern languages (Java is up to Version 7, Update 15, while C++ is now at C++11, with revisions planned ahead for C++14 and C++17). These changes, however, are evolutionary, not revolutionary. It is pretty much a guarantee that C++ code that runs today will still run with the latest version of the compiler in 2017. And no language is currently on the horizon to displace either Java or C++ from their dominant positions.

Instead, rather than develop newer and newer languages, we now extend our current software capabilities by writing support libraries and "importable" code (templates, generics) to extend the capabilities of our languages. We are adding additional tools (libraries) to support the compiler (another tool) to eventually/probably be converted to assembly language and then executed on the target machine.

Back in the 1970s and 1980s, the lack of the Internet made it difficult to share languages. Languages came into existence, were used for select projects, and disappeared in relative isolation. Languages tended to belong to a single project, or a single company. In the present, however, we can easily share languages and libraries. And because so many needed language features are common throughout much of the development community, new ideas for language features are easily and quickly shared. We can easily add standardized features (typically by including a new library or adding features to existing libraries) to languages that are standardized. Our extensibility is now managed by a mutually agreed upon standardization of languages. Rather than writing a new language, we have enough spare capability to add the libraries and compiler features to let the existing languages evolve.

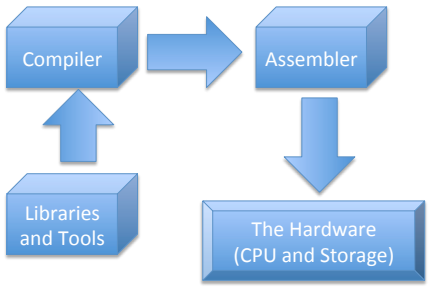


Figure 3 – using Libraries and Tools to further distance developers from the hardware

Coupled with the ability to "expand" languages through the use of libraries, we also have several other forces shaping how we develop software. These factors will have a tremendous effect on the software development of tomorrow.

The Near Future – Forces That Will Affect How We Develop Software

Distributed Computing

In the 1990s, we viewed distributed processing and parallel processing as the wave of the future. While both predictions have somewhat become true, it is not in a way that we ever envisioned 25 years ago. When CrossTalk first started publishing, 20 to 30 pound laptops were about as "portable" as computers could be. Back in the 1960s, when Star Trek first debuted, Star Fleet ensigns walked around the ship carrying PADDs, or Portable Access Display Devices. These devices, which seemed to be portable computers with access to the "Computer" were obviously a pipe dream. Now, as ultra books, full-fledged and high-powered laptops, smart phones and tablets abound we "distribute" computing and require software that equally distributes tasks as necessary. Mainstream languages now have extensions or specialized frameworks to allow developing software that runs on multiple platforms (from the large to the small).

Storage Issues

In the near future, several trends are going to affect how we develop software. The first is data storage. In the 1960s and 70s, the storage medium of choice was (as any addict of late-night really old science fiction movies can tell you) magnetic tape (for large data storage) and punched card (for individual programs). By the 1980s, floppy disks (8", 5 ¼" and later 3 ½") had become the medium of choice for individuals, while disk

storage was the standard for large data stores. By the 1990s, individual developers were using CD and DVDs for storage. By the 2000s, most developers had embraced flash storage with capacities up to 32GB being common. In all of the above examples, the devices for individual storage were "personal" under the total control of the developer. Now, however, cloud storage is becoming the standard. It is possible to obtain totally free cloud storage ranging from 5GB to 50GB. The side effect of this easy to obtain and easy to use (and extremely portable) storage is that the possession and protection of code and individual data is no longer under the developer's control.

Security Issues

Even before 9/11, military applications were routinely developed with a high level of security in the actual developed application. The events of 9/11 made security an integral part of almost all DoD system and development processes. With distributed computing (using smart phones, laptops and tables) and the use of cloud storage, DoD applications require specialized and higher levels of security during development. They also require a language (and operating system and network) that permits the applications to run with a relatively high degree of security.

In the 1980s and 1990s, software was developed mostly onsite, and typically run from a dedicated (and protected) client. Now, however, software development, execution of the applications, and code and data security are no longer necessarily centralized. When you combine the potential for terrorism and the potential for catastrophic failure of storage, applications will require unprecedented levels of security and redundancy. This has not been primarily a software issue in the past (it was handled by the operating system, network, and even manual processes). However, as redundancy and security will become more and more of a requirement for all levels of software in the future, I expect to see many security features become part of mainstream programming languages.

Trend To Graphical Languages or Graphical Front-Ends

Since the early 1950s, we have tried to use graphical methods to capture requirements and develop systems. We have tried flowcharts, State Transition Diagrams, Data Flow Diagrams, and the Unified Modeling Language. All work to help, but none are full-fledged enough to actually capture a full set of requirements for a large-scale system and produce executable code. Some (such as UML) come close.

In some areas, there do exist graphical interfaces that can create a complete executable system. For example, in the field of Modeling and Simulation, the language Arena (among others) allows an experienced user to capture requirements, develop the model, and execute the simulation under a variety of constraints [8].

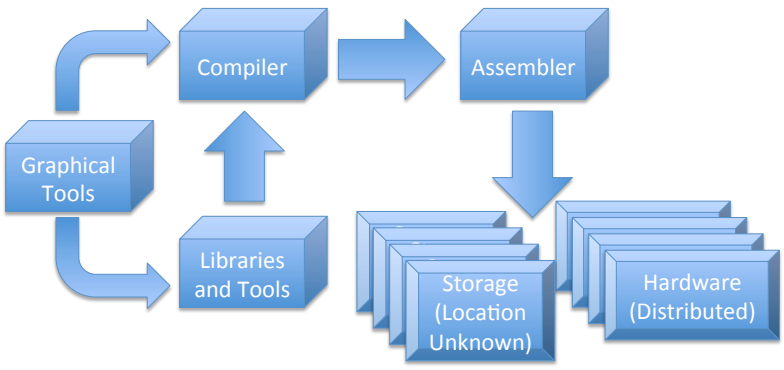


Figure 4 – Adding a Graphical Interface – even more removed from hardware, wherever it is, and wherever your storage is!

The Not-So-Near Future

Back in 1997, I was privileged to attend the ACM (Association of Computing Machinery) 50th Anniversary celebration, in San Jose. While there, a group of luminaries was present, and each was asked to briefly speak for 10 minutes or so on "What The Future Holds." I remember little about who spoke, or what they said, except for one speaker (whose name I cannot remember). He said, "10 years ago, we did not see the Internet coming, so who are we to predict the future?" I feel the same way. Things that we never envisioned as possible are now real. I can be standing in the middle of a cornfield in Nebraska, and given a decent 4G signal, have accessible to me almost all recorded history.

In the 1960s, when Star Trek had tablets disguised as PADDs, and cell phones and Bluetooth earpieces disguised as communicators, we could not comprehend a future with such wonderful devices. Now, I can wear a small device in my ear, tap it, and simply say, "Siri, please tell me the weather in London." I get results within seconds. The boundaries between normal life and computer usage are almost non-existent. Cars, appliances, even shoes are integrated in the ever-expanding computer-driven daily life.

I feel that software will continue to follow two separate paths—large-scale and non-traditional. Large-scale traditional software development (like much of the software developed within the DoD) will evolve slowly. Granted, I used Fortran in the 1960s, and now use C++, but the process is almost the same. Requirements, analysis, design, implementation, testing, maintenance—some things will probably not change for a long, long time. Niche software will come and go. A few new languages will be developed for specialized applications. It will be very difficult to create a new language that can overcome the developmental inertia that C++ and Java now hold. This language might continue to evolve (such as C# or Objective-C), but look at the staying power of Fortran. It was released commercially in 1957, and still maintains a strong "foot in the door" for many engineering applications. It appears than once a language becomes mainstream it remains a development tool for years and years to come.

Conclusions and Inescapable Facts

The average reader of CrossTalk is probably not the average developer of software. If you read CrossTalk, you probably work on large-scale or real-time systems. These systems are hard! We are always on the cutting edge of technology, trying to do what has never been done before.

I cannot say it any better than Fred Brooks said back in 1986, when he wrote the classic article, "No Silver Bullet—Essence and Accidents of Software Engineering [9]."

In it, he said, "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems."

If this is true, building software will always be hard. There is inherently no silver bullet.

Let us consider the inherent properties of this irreducible essence of modern software systems: complexity, conformity, changeability, and invisibility.

Twenty five years later, software is still hard. Software is still complex; still has to conform to bizarre and antiquated interfaces; still requires constant maintenance and updating; and still is essentially invisible, in spite of the graphical tools and process we try to use. And this is not necessarily bad. Using my iPhone to connect to a microprocessor in my shoes so that I can track my daily aerobic exercise history should be invisible—in fact, I want it seamless and thought-free.

But still, how do we create and provide this seamless integration between computers and every facet of our life? How about the really large-scale integration—the aircraft, spaceships, and weapons of tomorrow? Brooks, in the Mythical Man Month anniversary edition (where both the original article and his article "No Silver Bullet Refired" can be found) brings forth that perhaps methodologies are the silver bullet. The more advanced and larger the eventual software application, the more important it will be to have a process to manage the inherent complexity, conformity, changeability and invisibility.

And, as far as I can clearly see, therein lies the future. Processes are important—because of the magnitude of the effort. As the effort gets bigger, the more we need to rely on a process to guide us to completion. Back in the 1980s, when CrossTalk started publication, our computer systems were not exactly small, but they were smaller. For the mid 2010s? Double the CPU speed about 10 times. Then, also double available memory and storage capacity about the same number of times. And now fill up the computer with enough software to consume every clock cycle and byte. It is too big to even comprehend, so you better have a serious process to make it all fit together because without a process to manage the complexity you are not going to be able to get anything to work. In fact, you probably would not even be able to gather enough requirements to start development.

Large-scale projects require large-scale processes, which require relatively strict adherence to process standards. The languages we use are just a supporting role in the software systems we create. Software of the future is a combination of languages, tools, libraries, and most importantly, a process for putting it all together.

As we reflect on CrossTalk's 25 years of publication, I think that I can confidently say that CrossTalk has covered the issues and trends that got us to where we are now. As a frequent contributor and reviewer, I can also say that CrossTalk is already preparing us for the future!

ABOUT THE AUTHOR

David Cook is Associate Professor of Computer Science at Stephen F. Austin State University. He served 23 years in the Air Force, teaching computer science and software engineering at both the USAF Academy and AFIT. He also worked as a consultant to the STSC for 16 years. His fields of interest are software engineering, software quality, and verification and validation of large-scale modeling and simulations. His Ph.D. (in computer science) is from Texas A&M. He has been a columnist and contributing author for CrossTalk for almost all of its 25 years of publication.

E-mail: cookda@sfasu.edu
Phone: 936-468-2508

REFERENCES

1. Bellis, Mary. The History of the ENIAC Computer. February 24, 2013. <http://inventors.about.com/od/estartinventions/a/Eniac.htm>.
2. Leiner, Barry et. al. "www.internetsociety.org." October 2012. Brief History of the Internet. February 18, 2013. <http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet>.
3. Cook. "Evolution of Programming Languages and Why a Language is Not Enough to Solve Our Problems." Crosstalk, the Journal of Defense Software Engineering 12.12 (1999): 7-12.
4. Schorsch, Thomas and David Cook. "Evolutionary Trends in Programming Languages." CrossTalk, the Journal of Defense Software Engineering 16.2 (2003): 4-9.
5. Kinnersley, William. The Language List. 1991. February 13, 2013. <http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>.
6. Shankland, Stephen. Moore's Law: the rule that really matters in tech. October 2012. February 20, 2013. <http://news.cnet.com/8301-11386_3-57526581-76/moores-law-the-rule-that-really-matters-in-tech/>.
7. Venlet, VADM David. "Selected Acquisiton Report F-35." Department of Defense, n.d.
8. Kelton, David W. et. al. Simualtion with Arena. New York: McGraw-Hill, 2003.
9. Brooks, Frederick. Mythical Man Month, Anniversary Edition. Boston: Addison-Wesley, 1995.

A Twenty-Five Year Perspective

Karen Mercedes Goertzel, Booz Allen Hamilton

Abstract. The security risks associated with software and its development processes have been recognized for 40 years or more. But only in the past quarter century have efforts to understand and address the root causes of system security vulnerabilities evolved and coalesced into systematic efforts to improve software security assurance across government and leading industry sectors. Along with these programs have arisen efforts to reshape the software engineering profession, and to establish a robust software security technology and services industry.

This article provides a capsule history of the most significant of the software assurance efforts of the past 25 years, organized by the main problems they have striven—and continue to strive—to correct. At the end of the article, a number of more extensive, detailed software assurance landscapes are recommended to the reader, to complement and elaborate upon the information presented here.

Background

In 1974, a vulnerability analysis of the Multics multilevel secure operating system highlighted the potential of software design flaws and coding errors to be exploited as a means to compromise the security of the Multics system [1]. The report also discussed the potential for malicious insiders and external penetrators to exploit the lack of security awareness in Software Development Life Cycle (SDLC) processes and the absence of security protections for code development and distribution mechanisms to surreptitiously access and subvert the code prior to deployment. These process-level weaknesses were true not just for Multics, but for all of the software that made up the DoD's World Wide Military Command and Control System.

The 1974 report may be the first formal documentation of the direct correlations between (1) errors and flaws in a system's software and the vulnerability of that system, and (2) the lack of security controls in SDLC processes and the potential for malicious subversion of the software that results. However, the report's matter-of-fact tone suggests both problems were likely already well-recognized by then. And so the twin concerns that continue to drive virtually all software security assurance efforts to this day were already documented by 1974.

Over the next 20 years or so, any focus on improving software-level security assurance was limited to software-intensive systems with very high-confidence requirements used in the DoD, the Department of Energy, and the intelligence community (and in some of their non-U.S. counterparts abroad), e.g., the ballistic missile defense software developed under the Strategic Defense Initiative (SDI), and software used in high-assurance cryptographic systems, operating system kernels, and cross-domain solutions. Not until the mid-1990s did the broader security implications of the poor quality of most software explode into the broader consciousness. This awareness came thanks to the coincidence of the rise of universal Internet connectivity and the World Wide Web (and with it the exposure of increasing amounts of software that had previously operated only standalone or on private networks) with the global undertaking to examine and correct Y2K errors in the vast installed base of commercial and privately-produced software code. People were looking harder at their software than ever before, and what they found was not reassuring.

One result of the recognition that most software contained entirely too many exploitable errors and flaws was a deeper investigation into the root causes of the problem and, once identified, into the means to correct them. As a result, the late 1990s onward saw a growing ferment of commercial, academic, and government activity, including research, policy, process improvement, and propaganda—all falling under the rubric of "application security" or "software assurance."

By 2005, the President's Information Technology Advisory Committee was able to neatly summarize the twin security dilemmas that plague modern software:

"Today, as with cancer, vulnerable software can be invaded and modified to cause damage to previously healthy software, and infected software can replicate itself and be carried across networks to cause damage in other systems.... Vulnerabilities in software that are introduced by mistake or poor practices are a serious problem today. In the future, the Nation may face an even more challenging problem as adversaries—both foreign and domestic—become increasingly sophisticated in their ability to insert malicious code into critical software [4]."

The ability to exploit software's vulnerabilities to compromise its availability and the confidentiality and integrity of the information it handles, and the ability to exploit vulnerabilities in SDLC processes to intentionally subvert the functionality produced by those processes (by tampering with intended logic or implanting malicious code) continue to provide the impetus for all of today's software and software supply chain security assurance efforts. The rest of this article describes a representative sampling of such efforts over the past 25 years.

Exploitation of Software Vulnerabilities

The National Bureau of Standards published one of the first major taxonomies of operating system security vulnerabilities in 1976. Of the seven categories of vulnerabilities it identified, five constituted software vulnerabilities—(1) inadequate input/parameter validation, (2) incorrect input/parameter validation, (3) bounds checking errors, (4) race conditions, and (5) other exploitable logic errors [3]. The other two vulnerability categories were information flow-related.

By the mid-1990s, researchers recognized the need to clearly delineate software security from information security concerns, and several proposed taxonomies categorizing and characterizing software vulnerabilities were published by the Naval Research Laboratory, [5] Purdue University [6], the Open Web Application Security Project (OWASP) [7], and The MITRE Corporation [8].

In parallel with these efforts to "taxonomize" software vulnerabilities arose attempts to characterize techniques for exploiting software vulnerabilities, and tools and techniques such as attack trees [9], attack patterns [10], and threat modeling [11]

emerged to help developers characterize the attacks and exploitations that were most likely to target their software, and their likely outcomes.

Meanwhile, the typical timeframe between a vendor's discovery of a new vulnerability in its commercial software product and its ability to develop and release a fix, or "patch," to mitigate that vulnerability shrank from months to weeks to days to virtually nothing. This "zero-day vulnerability" problem, which had only been speculated about at the start of the millennium, was commonplace reality by the end of its first decade. In a struggle to maintain even tenuous control over the situation, both the software industry and the government began demanding exclusive rights to information about vulnerabilities discovered by their own and third-party "security researchers." In 2005, the first known sale of a vulnerability occurred. An ex-employee of the NSA sold information about an exploitable Linux flaw on an exclusive basis to the U.S. government; the alleged price: $80,000. Today, vendors routinely offer bounties for exclusive information about vulnerabilities in their products (Google reportedly spent upward of $460,000 in the first two years of its Vulnerability Reward Program). Buyers are motivated by the desire to keep news of vulnerabilities quiet long enough for patches to be released and applied, while many researchers seek to turn vulnerability-selling into a profitable industry. Some even market subscriptions to vendors whose software is affected. Others focus on the lucrative government market for vulnerabilities that can be exploited in information operations or cyber espionage [12].

Attacks targeting or exploiting software bugs, and the variety and capability of malicious logic have increased exponentially with the proliferation of network-connected software-intensive systems, services, and applications, including embedded systems. These risks plague not only the embedded software and microcode in military weapon systems, but in industrial control systems, networking devices, medical devices, onboard vehicle and avionic diagnostic systems, global positioning systems, mobile communications devices, consumer electronics, and an growing number of "smart" appliances in homes and workplaces. Many such systems are expected to operate continuously and cannot tolerate operational disruptions. A growing number are peripatetic, with no fixed location and only intermittent wireless connectivity. As a result, all are poor candidates for the traditional "push" approach to "just in time" software patching and updating.

At the same time, with miniaturization, hardware has also become so powerful that the lines between embedded software, firmware, and "fused-in" hardware logic have increasingly blurred. Researchers have also demonstrated the ability to load malicious firmware into information and communications technology (ICT) devices in order to subvert their operation. For example, a Columbia University research team installed malicious firmware in an HP LaserJet printer, then used it to illicitly forward documents from the print queue, and also to physically damage the printer [13]. Indeed, the problem of malicious firmware was explicitly documented by Scott Borg, the Director of the U.S. Cyber Consequences Unit and the Internet Security Alliance, in a 2008 strategy paper for the White House [14].

The emergence of post-manufacture reprogrammable integrated circuits in the 1990s obscured these distinctions even further, by expanding the threats to hardware logic beyond its fabrication and manufacturing processes. While all integrated circuits (ICs) are vulnerable to subversion during design and manufacture, field-programmable gate arrays (FPGAs) extend the attacker's window of opportunity, because their logic can be maliciously altered after manufacture. As long ago as 1999, researchers identified techniques for implanting, and resulting effects of, "FPGA viruses" [15], and demonstrated the ability to alter the bitstream used to reprogram the FPGA to insert malicious logic into its main memory [16]. A few years later, 2007, researchers at University of Illinois at Urbana-Champaign proved the feasibility of maliciously modifying non-reprogrammable IC logic to add post-deployment-exploitable "hardware Trojans" and "kill switches" [17].

In 2012, the Defense Advanced Research Projects Agency (DARPA) initiated its Vetting Commodity Information Technology (IT) Software and Firmware program "to look for innovative, large-scale approaches" for verifying that the software and firmware embedded in commodity IT devices purchased by DoD are "free of hidden backdoors and malicious functionality" [18]. In addition, software code analysis tool vendors such as Grammatech are expanding their products to support inspection of firmware for presence of vulnerabilities and malicious logic.

The need to expand the definition of "software" to include firmware and hardware logic reinforces the needs to also expand the focus of "software assurance" to address management of security risks in the supply chains for commercial software and hardware, as consumers—in DoD and beyond—continue to increase their reliance on COTS software, and reduce the amount of custom-development that allows them full lifecycle visibility and control over how their logic-bearing products are built and distributed.

Inadequate SDLC Processes and Technologies

In 1985, Canadian computer scientist David Lorge Parnas felt compelled to resign his position with the Strategic Defense Initiative Organization (SDIO) Panel on Computing in Support of Battle Management. In his letter of resignation he explained why he could no longer in good conscience associate himself with the SDI software development effort [19]. Given what was at stake—preventing a nuclear holocaust—SDI software could not afford to be less than 100 percent dependable. And 100 percent dependable software was (and still is) an impossibility. SDI software was so unprecedentedly huge and complex, Parnas explained, and its development methodology was so problematic, that any attempt to build assurably trustworthy SDI software was doomed to fail. Much of the fault lay in the limitations of conventional software development approaches—limitations that could not be overcome by the also-deficient emerging techniques of artificial intelligence, automatic programming, and formal methods. Parnas' letter provided the impetus for the SDIO to reconsider how its software would be developed. In 1990 two SDIO researchers published the Trusted Software Development Methodology (TSDM)—arguably the world's first secure SDLC methodology [20].

After TSDM, a number of "secure SDLC methodologies" were published. The most widely discussed of these is Microsoft's Trustworthy Computing Security Development Lifecycle (SDL) [21]. Others of note include John Viega's Comprehensive Light-

weight Application Security Process [22] and Gary McGraw's Seven Touch Points [23]. More recently, the BITS Financial Services Roundtable published a Software Assurance Framework [24]. In addition, a number of efforts have been undertaken to define a maturity model specific to software assurance processes; these include the software elements of the Systems Security Engineering Capability Maturity Model (SSE-CMM) [25], the Trusted Capability Maturity Model [26], the Federal Aviation Administration safety and security extensions to integrated capability maturity models [27], OWASP Open Software Assurance Maturity Model [28], and the Cigital/Fortify Building Security In Maturity Model [29].

The majority of SDLC security enhancements involve secure coding (also referred to as secure programming), the goal of which is to prevent avoidable code-level vulnerabilities, and security code review and software security testing, the goal of which is to detect design- and implementation-level vulnerabilities not avoided earlier in the SDLC. Secure coding requires inclusions of certain logic such as input validation of all parameters and explicit security-aware exception handling, avoidance of coding constructs and program calls associated with security vulnerabilities (e.g., printf in C and C++), use of type-safe and taintable programming languages, compilers that impose bounds checking, and "safe" libraries. Techniques for secure coding have been extensively documented in books, papers, and Web sites on the topic since the beginning of this century, and the Carnegie Mellon University Software Engineering Institute (CMU SEI) Computer Emergency Response Team's Secure Coding Initiative began publishing secure coding standards for C/C++ and Java in 2008 [30].

Other SDLC security enhancements have focused on protecting development artifacts both pre- and post-deployment. This includes secure software configuration management (SCM), with supporting secure SCM systems, and application of cryptographic integrity mechanisms to software executables prior to distribution, to name a few.

Subverted SDLC Processes and Malicious Logic

Outside of DoD, the primary motivation behind defining security-enhanced SDLC processes has been preventing avoidable but non-malicious vulnerabilities in software. But intentional subversion of software is a more potentially devastating problem. The shortcomings of SDLC processes for building DoD software, whether in the U.S. or offshore, and their exploitability to subvert or sabotage that software, have been repeatedly documented by the General Accountability Office [31].

Information on subversions by intentional malicious logic inclusions involving DoD or intelligence community software or developers is, unsurprisingly, virtually always classified. In other organizations, it also remains highly sensitive, for obvious reasons that if the SDLC vulnerabilities exploited and methods used to do so were widely known, they would provide other rogue developers (both inside and outside of software teams) with tried-and-true methods to copy. Because of this secrecy, it is difficult to provide examples of actual malicious code subversions. The fact that there is so much concern over the possibility is thought by many to prove the fact that such subversions have, in fact, occurred…and often. But coming up with unclassified examples is well-nigh impossible.

One of the most persistent examples has the dubious distinction of never having been authoritatively corroborated by any of the alleged participants. But it continues to stand as an "Emperor's New Clothes" type of object lesson, so it's worth mentioning here. The story goes that in 1982 a software time bomb was planted by agents of the U.S. Central Intelligence Agency in the software of a Canadian natural gas pipeline controller product. This subversion was performed in anticipation of that product falling into the hands of Soviet agents. The goal was to use the subverted software to sabotage the Trans Siberian gas pipeline (on which the controller was expected to be installed) in a manner so spectacular that it not only destroyed the pipeline, but also lead the Soviets to mistrust all the other sensitive Western technologies they had obtained through their industrial espionage program over the previous several years [32]. Less spectacular malware subversions in the private sector have led to prison terms for perpetrators such as Michael Don Skillern and Jeffrey Howard Gibson [33].

Given such examples (and, one suspects, many more in the classified literature), it is not surprising that prevention of subversion via malicious code has been at least as potent a driver for DoD's software assurance initiatives (and, more recently, its software supply chain risk management efforts) as avoiding software vulnerabilities. In 2007, NSA undertook a project to define guidelines focused specifically on adapting the SDLC to eliminate opportunities for pre-deployment malicious inclusions in software [34].

Non-functional Security Analysis and Testing of Software

Until the late 1980s, with the exception of code with very high confidence requirements (cryptographic code, multilevel secure trusted computing base code, etc.), security testing of software meant testing the functional correctness of software-implemented network-, system- and application-level security controls (e.g., authentication, access control, data encryption). If the software belonged to a system that handled classified data, some amount of penetration testing would be performed as part of system certification, focused on attempts to escalate privileges and inappropriately leak or steal sensitive data. Even the security analyses required for attaining higher levels of assurance under the Trusted Computer Security Evaluation Criteria and the Common Criteria focused on security function correctness and information flow vulnerabilities. To this day, the Common Criteria requires no security analysis to find exploitable code-level vulnerabilities or malicious logic.

One exception has been the expansion of fault tolerance, or resilience, testing to observation of executing code's behavior under the stressful conditions associated not only with unintentional faults but with intentional attempts to exploit software errors or induce failures (in the software itself, or the execution environment or infrastructure components on which it depends). Starting in the 1990s, researchers at University of Wisconsin at Madison took the lead in this kind of stress testing when they began a 20-year investigation into use of fuzzing as a means of testing software's ability to withstand denial of service attacks that targeted its weaknesses and exploited its flaws [35].

The 1990s also brought a growing awareness of software-level vulnerabilities in Web applications and other Web-facing

software. Publication of the OWASP Top 10 persuaded many software developers and buyers that they needed a cost-effective way to verify that their software could keep attackers out while still providing legitimate users with a conduit to the Web. The means they focused on were security code reviews (via static analysis to find known undesirable patterns in source code) and vulnerability scans (mainly of COTS software). Unfortunately, neither technique has proven very useful for detecting byzantine security faults or embedded malicious logic [36].

Software security testing techniques and tools over the past decade have vastly improved in terms of increased automation, improved accuracy with regard to minimizing "false positives" and "false negatives," and standardization and interoperability of outputs via efforts such as MITRE's Making Security Measurable and the future promise of software assurance ecosystems [37].

But while individual testing techniques and (semi)integrated software security testing toolsets have evolved quickly in sophistication and accuracy since the early 1990s [38], methodologies for software security testing are still rudimentary. There is still no software security counterpart of the network security integrated "situational awareness" view. Nor does there appear to be much research to conceive a "wholistic" strategy for choosing exactly the right combination of complementary techniques and tools to achieve maximally deep and comprehensive software security analysis and test coverage that remain flexible enough to adapt to the particular software technologies and program architecture of the test subject, are usable by testing teams of varying skills and knowledge, and feasible given varying available amounts of time and budget. Lack of such a strategic testing methodology means that anything more than automated vulnerability scanning remains too time consuming and costly for all but the most "critical" and "high confidence" software…the very software that, because it is considered critical or high confidence, is the most likely to have been engineered with caution under controlled conditions, and is therefore in less need of extensive security testing.

Software Intellectual Property: Piracy, Theft, and Tampering

From the 1980s onward the single greatest "security" concern of software vendors has been the protection of their intellectual property (IP). DoD too is concerned with protecting software IP, though for different reasons.

Vendors' main concern has been piracy—the unauthorized copying and distribution of licensed software. DoD, on the other hand, is most concerned about adversaries gaining access to the IP inherent in source code of their critical software, either via reverse engineering from binaries or direct source code theft, as a step towards producing tampered, malicious versions, or studying its operation and vulnerabilities to better target or counter the systems in which it is used (e.g., weapon systems), or to obtain code on which to base comparable capabilities for their own use (in essence, piracy).

Piracy is a major concern to vendors because of the revenue loss it represents. In the 1980s, dozens of vendors rushed out hardware "dongles" for mandatory co-installation on computers on which their software was installed. The dongles ensured that the software could run only on the system for which it was licensed, and to which dongle was attached. This meant the code would not operate if copied to another system. The problem was that enterprise users had a legitimate need for backup copies of software as part of continuity of operations planning. And like any other small item, dongles were easy to misplace. So in the face of customer complaints, by the mid-1990s, most vendors had abandoned the devices in favor of digital rights management controls that accomplished essentially the same protections, and is still used by many software vendors today [39]. Over the past decade, the software industry has launched numerous anti-piracy initiatives and campaigns, individually and via their industry trade associations [40].

Protection against executable software reverse engineering led DoD, in December 2001, to establish its Software Protection Initiative (SPI). SPI develops and deploys intellectual property protections within national security system software to prevent post-deployment reverse engineering and reconnaissance, misuse, and abuse by adversaries [41]. Since its inception, the SPI has sponsored much of the significant research and development of technologies for software IP protection (e.g., anti-reverse engineering), software integrity protection (e.g., tamper-proofing), and software anti-counterfeiting.

Preventing source code theft is a problem for both vendors and government software projects. It requires both secure configuration management and effective cybersecurity protections for the computing and networking infrastructure relied on by software teams. Google discovered this to its great consternation in 2009, when Internet-based intruders stole the source code of the password management system used in most Google Web services, including Gmail. The method by which the intruders got access to the code reads like Web Application Insecurity 101: a Google China employee clicked on a link in a Microsoft Messenger message that redirected him to a malicious Web site. From there, the intruders accessed and took control of his computer, and a few short hops later, found and took control of the software repository in which the development team at Google headquarters stored the password management system code [42].

Doing Something About It: Software Assurance Initiatives and Public-Private Partnerships

In 1998 Microsoft, the world's largest software vendor, could no longer keep up with the exponential increase in reported vulnerabilities in its operating system and Web products. The company set up an internal security task force to investigate the vulnerabilities' root causes, then following the task force's recommendations, established product line security initiatives and "pushes" from 1999-2004 that ultimately coalesced into the Microsoft Trustworthy Software Development program. Two significant artifacts of the program were mandated company-wide and widely published for adoption by third-party suppliers to Microsoft (and anyone else who cared to adopt of them)—the "STRIDE/DREAD" threat modeling methodology and the SDL methodology. While other software vendors also adopted software assurance measures in the same timeframe (e.g., by the early 2000s Oracle Corporation had committed to a fairly rigorous software assurance regime), few of the others were as forthcoming or influential as Microsoft.

Seeing the world's leading software vendor change its *modus operandi* in so public a manner was an important factor in

increasing software buyers' awareness of the need for more secure software products. The OWASP Top 10 was another. Soon, organizations were rushing to discover whether their Web applications harbored any of the Top 10—and to demand that their software vendors do the same. This engendered a new industry of semi-automated tools for static security analysis of source code, and automated scanners for finding vulnerabilities in (mainly Web) application executables.

In the mid-2000s, consortia software vendors (often led by Microsoft), software security tool vendors, and corporate software users seemed to spring up every few months, including the Web Application Security Consortium in 2004, the Secure Software Forum and Application Security Industry Consortium in 2005, and SAFECode in 2007. 2007 also saw Concurrent Technologies Corp. announce the short-lived Software Assurance Consortium.

The financial services sector has also been active in its pursuit and promotion of software assurance in the context of payment and banking application security. The Visa USA Cardholder Information Security Program Payment Application Best Practices expanded and evolved into the Payment Card Industry Security Standards Council's Application Data Security Standard, now a de facto standard across the financial services industry worldwide. In the U.S., the BITS Financial Services Roundtable has undertaken a Software Security and Patch Management Initiative and Product Certification Program and produced a Software Security and Patch Management Toolkit and Software Assurance Framework for use by its members and the broader financial services community.

In the public sector, the Defense Information Systems Agency (DISA) may have been the first since the SDIO to take on the challenge of identifying and promoting methods, techniques, and supporting tools for secure software development. The three-year Application Security Project began in 2002 as a means of reducing the likelihood of OWASP Top 10 vulnerabilities in DoD Web technology-based application systems. The project's broad agenda included (1) producing developer guidance based on recognized full-SDLC best practices for secure application development; (2) assembling a portable, automated application security testing toolkit and supporting methodology with which it could offer an application vulnerability assessment service to DoD software programs; (3) defining a "reference set" of security requirements for DoD developers to leverage in their application specifications. By the end of 2004, however, DISA shifted its focus away from attempts to proactively improve the processes by which DoD software was built to reactively assessing the security of DoD software. This shift was reflected in the move of the Project to DISA's Field Security Operation, which reinterpreted the content of the Project's deliverables into a single Application Security and Development Security Technical Implementation Guide (STIG) [43] and supporting checklist. It was left up to software project managers to figure out how to ensure their teams developed software that could pass the STIG checks.

Elsewhere in DoD, security of mission critical software and risks posed by the increasing offshoring of that software were driving new initiatives. In December 1999, the Defense Science Board (DSB) suggested that the Assistant Secretary of Defense (ASD) for Command, Control, Communications, and Intelligence "develop and promulgate an Essential System Software Assurance Program" [44]. It took the ASD for Networks and Information Integration (NII) nearly four years to do just that: In June 2003, the DoD Software Assurance Initiative undertook to establish methods for evaluating and measuring assurance risks associated with commercial software, including accurate detection of the software's pedigree and provenance. In 2004, ASD(NII) joined with the Office of the Under Secretary of Defense for Acquisition, Technology and Logistics to form a Software Assurance Tiger Team for strategizing how DoD and broader Federal government would reduce its exposure to software assurance risks. The Tiger Team enlisted industry partners via the National Defense Industrial Association (NDIA), Aerospace Industries Association, Government Electronics and Information Technology Association, and Object Management Group.

The Software Assurance Initiative soon reached broad consensus on the impracticality of relying on pedigree and provenance to justify confidence in acquired software. This triggered a shift in their philosophy: all commercial software was to be considered potentially vulnerable and malicious, and engineering techniques had to be adopted to render DoD systems resilient against its destructive effects. Thus, the Initiative recast itself as the DoD System Assurance Program and, with the assistance of NDIA, developed *Engineering for System Assurance* [45] (1st edition, 2006; 2nd edition, 2008), which was expanded and adopted as a NATO engineering standard in 2010 [46].

In 2005, NSA established its Center for Assured Software (CAS) as the focal point for software assurance issues in the defense intelligence community (and in broader DoD). CAS collaborates closely with the DHS/DoD/NIST co-sponsored Software Assurance working groups and fora. CAS also influences, and in some case leads, development of DoD software assurance-related standards and policy, research, and evaluation processes, and strives to push the state of the art in software analysis tools and assessment methods. In 2009, the CAS undertook an Assurance Development Processes strategic initiative to establish trustworthy best-practice-based software development processes across DoD and the intelligence community. For several years, NSA also ran a Code Assessment Methodology Project to evaluate the security of source code to be used in high-assurance, critical DoD systems.

More recently, DoD's software assurance concerns have turned to the problems of securing the software supply chain, as a component of the larger Comprehensive National Cybersecurity Initiative (CNCI) ICT Supply Chain Risk Management (SCRM) Initiative 11, which is described—together with broader DoD and other Federal government ICT SCRM activities and programs—in the DoD Information Assurance Technology Analysis Center (IATAC) 2009 state of the art report on ICT SCRM [47].

In 2003, in parallel with DoD's efforts, DHS was assigned responsibility for responding to the *National Strategy to Secure Cyberspace*'s call for establishment of a national program to "reduce and remediate software vulnerabilities" and for facilitating "a national public-private effort to promulgate best practices and methodologies that promote integrity, security, and reliability in software code development, including processes and procedures that diminish the possibilities of erroneous code, malicious code, or trap doors" being introduced into code under

development. These responsibilities led, a year later, to the DHS Software Assurance Program. Coordinated with and complementing the efforts of DoD and the NIST Software Assurance Metrics and Tools Evaluation program (largely funded by DHS), the main thrusts of the DHS Program have been to publish secure SDLC information and guidance, promote security in software practitioner education and training, software assurance professional certification, and standardization of software assurance-related taxonomies, tool outputs, and metrics, as well as general awareness-raising. As with DoD, the Program also more recently shifted its focus to address security risks in the commercial and open source software supply chains.

Taking on the DSB's challenge that computer science academic curricula were "inadequate in terms of stressing practices for quality and security, or inculcating developers with a defensive mindset" [48], DHS made software assurance education and training one of its key thrusts from its inception. In 2006 it published a software assurance "common body of knowledge" for use in developing university curricula and courseware [49]. By 2010, the Program could boast of the Institute of Electrical and Electronics Engineers Computer Society's recognition of the Master of Software Assurance Reference Curriculum collaboratively developed by researchers and educators at several universities under DHS sponsorship [50]. IATAC's 2007 state of the report, *Software Security Assurance* [51] lists numerous examples of universities with dedicated graduate-level teaching of software assurance, advanced degrees in software assurance-related disciplines, software security research projects and labs, as well as professional training vendors with secure software development offerings, and emerging (now established) professional certifications for developers and project managers in the discipline of software security assurance and secure programming—most notably the Certified Software Security Lifecycle Professional administered by the International Information Systems Security Certification Consortium and the SANS Software Security Institute's Secure Programming Skills Assessment and Certified Application Security Professional certification. While the education/training and certification landscape described in the IATAC report has shifted somewhat in the subsequent six years, it remains generally representative.

Unlike DoD System Assurance's limited public partnerships, DHS's outreach is literally global, encompassing U.S. federal, state, and local and allied government users and producers of software, software and software security tool vendors, and academia [52]. The Program's main outreach mechanisms are its semi-annual Software Assurance Forums and more-frequent working group meetings (co-sponsored with DoD and NIST). The driving philosophy behind DHS efforts is that a general move towards more secure software worldwide will benefit federal government and DHS-protected infrastructure sectors in particular. DoD benefits from DHS's more global approach through active co-sponsorship of and participation in DHS-spearheaded endeavors. The efforts of DoD and DHS have also inspired comparable undertakings by allied governments. For example, in 2011, the United Kingdom established its own Trustworthy Software Initiative in response to 2010's *National Security Strategy of Cybersecurity* [53], which identified lack of secure, dependable, resilient software as a critical risk to the UK's cybersecurity posture. And NATO published *Engineering Methods and Tools for Software Safety and Security* in 2009 [54].

A number of significant software assurance research initiatives are ongoing in the U.S. and abroad, especially in Europe, including the Network of Excellence on Engineering Secure Future Internet Software Services and Systems project sponsored by the European Commission's Seventh Framework Programme for Research (FP7) [53].

Conclusion

It has often been claimed that we already know how to build secure software. If this is true, why don't we just do it? But no matter how much lip service they pay to wanting software that has fewer vulnerabilities and is less susceptible to malicious inclusions, most suppliers and consumers still make their how-to-build and what-to-buy decisions based on cost, or on a "value proposition" that boils down to how fast innovations can be turned into available product, and cost. Few buyers are willing to wait longer and pay more so software can undergo the disciplined engineering needed to assure its trustworthiness and dependability. Nor are they willing to forego desirable innovations just because the level of security risk they pose is not (and possibly cannot) be known. Nor will suppliers willingly invest in and enforce software assurance measures that few customers demand.

But the continuing, and indeed growing, reliance on COTS and open source software means that, to succeed in the long term, software assurance efforts cannot remain limited to the small subset of software deemed "high consequence" or "trusted." It is impossible to predict which of today's "general purpose" software products will end up in tomorrow's high consequence, trusted systems, just as it was impossible to predict in 1988 that Microsoft Excel and Internet Explorer (to name two examples) would, in spite of their persistent, myriad vulnerabilities and susceptibility to malicious insertions, emerge as vital components of mission critical national security systems.

Instead of attempting to second guess which software needs to be trustworthy and dependable, software assurance should be applied systematically and comprehensively to all software. For this to happen, future software assurance efforts need to finally take on the elephant in the room: the need to change the psychology of the suppliers and consumers. Awareness campaigns and polite suggestions of software assurance content for post-graduate academic software engineering curricula attempt to persuade and reeducate developers and consumers long after their bad habits have been formed, and are thus far too little far too late. A "software assurance mentality" needs to be inculcated during the very earliest years in which future developers and users encounter software and begin to understand how it works and its value to them. Software (and machine logic) are nearly universal today, and are only going to become more integral to every aspect of daily life by the time the next generation of developers and users reach working age. For this reason, tomorrow's developers and users need to be taught from early childhood that threats to the security of software and logic-bearing devices are threats to their own personal privacy, health, safety, financial security, and, ultimately, happiness.

Other Attempts to Characterize the Software Assurance Landscape

This article has only touched on highlights of the last 25 years of software security assurance initiatives and trends. There have been several earlier efforts to depict the software assurance landscape in varying levels of detail. Interested readers are encouraged to take a look at:

- Goertzel, Karen Mercedes, et al. *Software Security Assurance* (see reference 51).
- Davis, Noopur. *Secure Software Development Life Cycle Processes: A Technology Scouting Report*. Technical Note CMU/SEI-2005-TN-024, December 2005 <http://www.sei.cmu.edu/reports/05tn024.pdf>
- Jayaram, K.R., and Aditya P. Mathur. *Software Engineering for Secure Software–State of the Art: A Survey*. Purdue University Center for Education and Research in Information Assurance and Security and Software Engineering Research Center Technical Report 2005-67, 19 September 2005 <http://www.cerias.purdue.edu/assets/pdf/bibtex_archive/2005-67.pdf>
- DHS. *Software Assurance Landscape*. Preliminary Draft, 28 August 2006 <https://www.owasp.org/images/6/6c/Software_Assurance_Landscape_-_Preliminary_Draft_1.doc>
- Graff, Mark D. *Secure Coding: The State of the Practice*, 2001 <http://markgraff.com/mg_writings/SC_2001_public.pdf>
- Essafi, Mehrez, et al. *Towards a Comprehensive View of Secure Software Engineering. Proceedings of the International Conference on Emerging Security Information, Systems, and Technologies (SecureWare 2007)*, Valencia, Spain, 14-20 October 2007 <http://www.researchgate.net/publication/4292729_Towards_a_Comprehensive_View_of_Secure_Software_Engineering/file/79e4150bb0ce96e522.pdf>

ABOUT THE AUTHOR

Karen Mercedes Goertzel, CISSP, is an expert in application security and software and hardware assurance, the insider threat to information systems, assured information sharing, emerging cybersecurity technologies, and ICT supply chain risk management. She has performed in-depth research and analysis and policy and guidance development for customers in the U.S. financial and ICT industries, DoD, the intelligence community, Department of State, NIST, IRS, and other civilian government agencies in the U.S., the UK, NATO, Australia, and Canada.

7710 Random Run Lane–Suite 103
Falls Church, VA 22042-7769
703-698-7454
goertzel_karen@bah.com

ADDITIONAL SUGGESTED READING

(Contact the author to request a comprehensive list of published books on software security assurance.)

1. DHS/US-CERT. Build Security In Web portal. <https://buildsecurityin.us-cert.gov>
2. David A. Wheeler's Personal Home Page. <http://www.dwheeler.com>
3. CMU SEI. *Insider Threats in the Software Development Lifecycle*, 23 February 2011 <http://www.cert.org/archive/pdf/sepg500.pdf>
4. Fedchak, Elaine, et al. *Software Project Management for Software Assurance*. Data and Analysis Center for Software (DACS) Report Number 347617 (Rome, NY: DACS, 30 September 2007) <http://www.thedacs.com/get_pdf/DACS-347617.pdf>
5. *International Journal of Secure Software Engineering* <http://www.igi-global.com/journal/international-journal-secure-software-engineering/1159>

REFERENCES

1. Karger, Paul A., and Roger R. Schell, *Multics Security Evaluation: Vulnerability Analysis*. U.S. Air Force Electronics Systems Division Report ESD-TR-74-193, Volume II, June 1974. <http://seclab.cs.ucdavis.edu/projects/history/papers/karg74.pdf>
2. Abbot, Robert P., *The RISOS [Research into Secure Operating Systems] Project: Security Analysis and Enhancements of Computer Operating Systems*. National Bureau of Standards Interagency Report Number NBSIR 76-1041, 1976.
3. The Karger/Schell Multics analysis in 1974 had also identified buffer and stack overflow errors and lack of adequate input validation as the system's most significant, proven-exploitable vulnerabilities. More than a decade later, in 1988, a buffer overflow in the Berkeley Unix finger daemon was exploited by Robert Tappan Morris to launch the world's first Internet worm. 25 years on, the 2011 Veracode *State of Software Security Report* found that exploitable buffer overflows and memory management issues were still the most prevalent vulnerabilities in commercial software. *Plus ça change, plus c'est la même chose.*
4. PITAC. *Report to the President on Cyber Security: A Crisis of Prioritization*, February 2005. <http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf>
5. Landwehr, Carl E., et al. "A Taxonomy of Computer Program Security Flaws, with Examples". Naval Research Laboratory Center for High Assurance Computer Systems technical report NRL/FR/5542-93-9591, 19 November 1993. <http://chacs.nrl.navy.mil/publications/CHACS/1994/1994landwehr-acmcs.pdf;>
6. Du, Wenliang, and Aditya Mathur, "Categorization of Software Errors That Led to Security Breaches". *Proceedings of the National Information Systems Security Conference*, 1998. <http://www.cis.syr.edu/~wedu/Research/paper/nissc98.ps;>
7. Open Web Application Security Project (OWASP). "Top Ten Most Critical Web Application Security Vulnerabilities", 2002 (and revised several times since then). <https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project>
8. The Common Weakness Enumeration (CWE) <http://cwe.mitre.org/> is a standard dictionary of "root causes" at the specification, design, and implementation levels of exploitable software security vulnerabilities, such as those listed (along with system and network vulnerabilities) in MITRE's Common Vulnerabilities and Exposures (CVE; <http://cve.mitre.org>). In 2012 CWE was adopted by the International Telecommunications Union (ITU) as *Recommendation X.1524 : Common weakness enumeration.* <http://www.itu.int/rec/T-REC-X.1524-201203-I/en>
9. Schneier, Bruce. "Attack Trees". Dr. Dobb's Journal, December 1999. <http://www.schneier.com/paper-attacktrees-ddj-ft.html>
10. Moore, Andrew P., et al. "Attack Modeling for Information Security and Survivability". CMU/SEI-2001-TN-001, March 2001. <http://www.cert.org/archive/pdf/01tn001.pdf>. The culmination of attack pattern-definition attempts is MITRE's Common Attack Pattern Enumeration and Classification (see <http://capec.mitre.org/>), a standardized list of the most common techniques for exploiting the vulnerabilities that can result from CWEs.
11. Howard, Michael, and David LeBlanc. Chapter 2, "Security Design by Threat Modeling". *Writing Secure Code*. (Redmond, WA: Microsoft Press, 2002).
12. Gonsalves, Antone. "The Shadowy World of Selling Software Bugs, and How It Makes Us all Less Safe". *readwrite hack*, 4 October 2012. <http://readwrite.com/2012/10/04/the-shadowy-world-of-selling-software-bugs-and-how-it-makes-us-all-less-safe>
13. Rashid, Fahmida. "Researchers Hijack Printer Using Malicious Firmware Update". *eWeek*, 29 November 2011.

14. Borg, Scott. "Securing the Supply Chain for Electronic Equipment: A Strategy and Framework". Whitepaper developed for the White House by the Internet Security Alliance, November 2008.
15. Hadži, Ilija, et al. "FPGA Viruses". University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-99-06, January 1999. <http://www.cis.upenn.edu/~jms/papers/fpgavirus.pdf>
16. Torres, Lionel, et al. "Security and FPGA: Analysis and Trends". Deliverable SP1 for the ICT for Emerging Regions Project of France's Agence Nationale de Recherche, 31 January 2007. < http://www.lirmm.fr/~w3mic/ANR/PDF/D1.pdf >
17. King, Samuel T., et al. "Designing and Implementing Malicious Hardware". *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats*. San Francisco, CA, 15 April 2008. < http://static.usenix.org/events/leet08/tech/full_papers/king/king.pdf http://www.whitehouse.gov/files/documents/cyber/ISA - Securing the Supply Chain for Electronic Equipment.pdf>
18. DARPA Press Release. "New DARPA Program Seeks to Reveal Backdoors and Other Hidden Malicious Functionality in Commercial IT Devices". 30 November 2012.
19. Later published under the title "Software Aspects of Strategic Defense Systems". Communications of the ACM [Association for Computing Machinery], Vol. 28 No. 12, December 1985. <http://klabs.org/richcontent/software_content/papers/parnas_acm_85.pdf>
20. Watson, John, and Edward Amoroso. "A Trusted Software Development Methodology". *Proceedings of the 13th National Computer Security Conference*, Volume II. Washington, D.C., 1990. TSDM was later renamed Trusted Software Methodology.
21. Lipner, Steven B. "The Trustworthy Computing Security Development Lifecycle". Proceedings of the 20th Annual Computer Security Applications Conference, Tucson, AZ, 6-10 December 2004.
22. Viega, John. "Security in the Software Development Lifecycle". IBM developerWorks, 15 October 2004. <http://www.ibm.com/developerworks/rational/library/content/RationalEdge/oct04/viega/viega.pdf>
23. McGraw, Gary. "The Seven Touch Points of Secure Software". Dr. Dobb's Journal, 1 September 2005. <http://www.drdobbs.com/the-7-touchpoints-of-secure-software/184415391>
24. BITS Financial Services Roundtable. *Software Assurance Framework*. January 2012. <http://www.bits.org/publications/security/BITSSoftwareAssurance0112.pdf>
25. SSE-CMM Project. *Systems Security Engineering Capability Maturity Model*, Version 1.0. 21 October 1996.
26. DHS. Section D.5. *Security in the Software Lifecycle*, Draft Version 1.2, August 2006. <http://www.cert.org/books/secureswe/SecuritySL.pdf> Note that this book was significantly revised and republished by DHS in 2008 as Enhancing the Development Lifecycle to Produce Secure Software.
27. Ibrahim, Linda, et al. "Safety and Security Extensions for Integrated Capability Maturity Models", September 2004. <http://www.faa.gov/about/office_org/headquarters_offices/aio/library/media/SafetyandSecurityExt-FINAL-web.pdf>
28. Version 1 of OpenSAMM was released in 2008. See OWASP's OpenSAMM Web pages and the OpenSAMM Web site. <http://www.opensamm.org> <https://www.owasp.org/index.php/Category:Software_Assurance_Maturity_Model>
29. Version 1 of the BSIMM was published in 2009. The current version is Version 4. See the BSIMM Web site. <http://bsimm.com/>
30. CMU SEI Secure Coding Standards Web page. <http://www.cert.org/secure-coding/scstandards.html>
31. Government Accountability Office (GAO). *DoD Information Security: Serious Weaknesses Continue to Place Defense Operations at Risk*. GAO/AIMD-99-107, August 1999. <http://www.gao.gov/products/AIMD-99-107>. GAO. *Defense Acquisitions: Knowledge of Software Suppliers Needed to Manage Risk*. GAO-04-678, May 2004. <http://www.gao.gov/new.items/d04678.pdf>. GAO. *Offshoring of Services: an Overview of the Issues*. GAO-06-5, November 2005. <http://www.gao.gov/new.items/d065.pdf>
32. Murdoch, Steven. "Destructive Activism: The Double-Edged Sword of Digital Tactics". Joyce, Mary, editor, *Digital Activism Decoded: The New Mechanics of Change* (New York, NY: International Debate Education Association, 2010). Also Weiss, Gus W., "Duping the Soviets: The Farewell Dossier". *Studies in Intelligence*, Volume 29 Number 5, 1996. <https://www.cia.gov/library/center-for-the-study-of-intelligence/kent-csi/vol39no5/pdf/v39i5a14p.pdf>
33. Gabrielson, Bruce, Karen Mercedes Goertzel, et al. Appendix E, "Real World Insider Abuse Cases". *The Insider Threat to Information Systems* [Unclassified, For Official Use Only, U.S. Government and Contractors Only]. (Herndon, VA: IATAC, 10 October 2008).
34. NSA. *Guidance for Addressing Malicious Code Risk*, 10 September 2007. <http://www.nsa.gov/ia/_files/Guidance_For_Addressing_Malicious_Code_Risk.pdf>
35. Miller, Barton P., et al. "An Empirical Study of the Reliability of UNIX Utilities". *Communications of the ACM*, Volume 33 Number 12, December 1990. <ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf>; University of Wisconsin-Madison Fuzz Testing of Application Reliability Web page. <http://pages.cs.wisc.edu/~bart/fuzz/fuzz.html>
36. The annual Underhanded C Code Contest is designed to illustrate how ineffective code reviews are for finding malicious inclusions in code. Contest submissions must be malicious while maintaining "plausible deniability", i.e., their malicious logic must by definition not be detectable through static analysis. See <http://underhanded.xcott.com/>
37. Such as KDM Analytics' Software Assurance Ecosystem, <http://www.kdmanalytics.com/swa/ecosystem.php>, and Object Management Group's Software Assurance Ecosystem, < http://sysa.omg.org/docs/SwA Ecosystem/Assurance Ecosystem.ppt>.
38. Information on the wide range of tools available can be found on the DHS Build Security In Web portal, <https://buildsecurityin.us-cert.gov/>, the NIST Software Assurance Metrics and Tools Evaluation portal, <http://samate.nist.gov/>, and in Goertzel, Karen Mercedes, et al. *Vulnerability Assessment* (Herndon, VA: IATAC, May 2011), <http://iac.dtic.mil/iatac/download/vulnerability_assessment.pdf>. Outsourcing to expert service providers such as Veracode and Aspect Security may be an alternative to doing one's own security analyses and tests.
39. Belovich, Steve, Ph.D. "IT Security History and Architecture–Part 5 of 6". Infosec Island, 24 August 2010. <http://infosecisland.com/blogview/6931-IT-Security-History-and-Architecture-Part-5-of-6.html>
40. The most noteworthy are those of the Business Software Alliance, Software and Information Industry Association, Entertainment and Leisure Software Publishers Association, Information Technology and Innovation Foundation, Federation Against Software Theft, Entertainment Software Association, Content Delivery and Storage Association, and the Association for Copyright of Computer Software. See: Section 4.7.27, "Anti-Piracy Initiatives". Goertzel, Karen Mercedes, et al. *Security Risk Management in the Off-the-Shelf (OTS) Information and Communications Technology (ICT) Supply Chain: a State-of-the-Art Report* [U.S. Government and Contractors Only] (Herndon, VA: IATAC, 17 August 2010).
41. Hughes, Jeff, and Martin R. Stytz. "Advancing Software Security: the Software Protection Initiative". <http://www.preemptive.com/documentation/SPI_software_Protection_Initative.pdf>
42. Markoff, John. "Cyberattack on Google Said to Hit Password System". *The New York Times*, 19 April 2010. <http://www.nytimes.com/2010/04/20/technology/20google.html>
43. DISA. *Application Security and Development STIG*. <http://iase.disa.mil/stigs/app_security/app_sec/app_sec.html>
44. Defense Science Board (DSB). *Final Report of the Defense Science Board Task Force on Globalization and Security* (especially Annex IV), December 1999. <http://www.acq.osd.mil/dsb/reports/globalization.pdf>
45. DoD System Assurance Working Group and NDIA. *Engineering for System Assurance*, Version 1.0, 2008. <http://www.acq.osd.mil/sse/docs/SA-Guidebook-v1-Oct2008.pdf>
46. NATO Standard AEP-67. *Engineering for System Assurance in NATO Programmes*, First Edition, 4 February 2010 <http://nsa.nato.int/nsa/zpublic/ap/aep-67(1)e.pdf>
47. Op. cit. Goertzel, et al. *Security Risk Management in the OTS ICT Supply Chain*.
48. DSB. *Final Report of the Task Force on Mission Impact of Foreign Influence on DoD Software*, September 2007. <http://www.cyber.st.dhs.gov/docs/Defense Science Board Task Force–Report on Mission Impact of Foreign Influence on DoD Software (2007).pdf>
49. DHS Software Assurance Common Body of Knowledge/Principles Organization Web page. <https://buildsecurityin.us-cert.gov/bsi/dhs/927-BSI.html>
50. IEEE Computer Society. "Computer Society Recognizes Master of Software Assurance Curriculum". Press release dated 8 December 2010. <http://www.computer.org/portal/web/pressroom/20101213MSWA> and CMU SEI Software Assurance Curriculum Web page. <http://www.cert.org/mswa/> Also DHS Software Assurance Curriculum Project Web page. <https://buildsecurityin.us-cert.gov/bsi/1165-BSI.html>
51. Goertzel, Karen Mercedes, et al. *Software Security Assurance* (Herndon, VA: IATAC, 31 July 2007). <http://iac.dtic.mil/csiac/download/security.pdf>. Note that in April/May 2013, DoD tasked the Defense Technical Information Center's Cyber Security Information Analysis Center (the successor to IATAC) to produce an update of this report, which is expected to be published in late 2013.
52. A number of other defense and civil agencies are also committed to improving software security assurance. For example, Part 10 Chapter 8 of the *Internal Revenue Service Manual* includes a section entitled "Secure Application Development". <http://www.irs.gov/irm/part10/irm_10-008-006.html>
53. Her Majesty's Government Cabinet Office. *Cyber Security Strategy*, 2010 (revised November 2011). <http://www.cabinetoffice.gov.uk/resource-library/cyber-security-strategy>
54. Broy, Manfred, et al., editors. *Engineering Methods and Tools for Software Safety and Security*. NATO Science for Peace and Security Series D: Information and Communication Security, Volume 22 (Amsterdam, The Netherlands: IOS Press, 2009).
55. Network of Excellence on Engineering Secure Future Internet Software Services and Systems Web page. <http://www.nessos-project.eu/>

Is Something Missing From Project Management?

Walt Lipke, PMI - Oklahoma City Chapter

Abstract. There are many elements to a project … requirements, schedule, cost, quality, human resources, communications, risk, procurement, and… Every project is complex and extremely difficult to manage to successful completion, even those considered "small." The majority of the life of a project occurs during its execution. Although the execution phase is preponderant, there does not seem to be much emphasis on it. The literature, the training, professional meetings, and conferences do not commit proportionate energy to methods and techniques to prepare project managers for monitoring and reporting performance. Neither do these venues for knowledge transference bring focus to addressing performance measures and indicators, or using them for controlling the project. This paper examines the assertion and proposes the application of earned value management and its extension, earned schedule, as a way forward.[1]

Introduction

Over the last 30 years, from about 1980 until the present, there has been a significant evolution in software development, quality systems and project management. The foundation for this advancement in practice is strongly connected to a few devoted quality experts and world events occurring more than 70 years ago.

After World War II the U.S. was the predominant industrial nation in the world. The U.S. produced. The world consumed. The quality of the U.S. products was of little concern; they would sell regardless. This economic position was held until about 1970 after which the market for U.S. products declined.

Beginning with the post war reconstruction, Japan's business leaders learned and adopted manufacturing practices the U.S. utilized during and prior to WWII. Most notably, the Japanese were taught the methods of quality by W. Edwards Deming. As Deming had prophesied to Japan's leaders, economic growth came from their dedicated use of the techniques he had learned from Walter Shewhart at Bell Laboratories.

During the 1980s Japan's automobile industry began to make noticeable inroads into the U.S. market. Their success was an alarming wake-up to U.S. manufacturers, who recognized that they truly had serious competition. Thus began the quality revolution in the United States.

No longer was quality perceived as an expendable portion of the production process, largely ignored. During this period, Deming videos and seminars were commonplace. Every industry was determined to improve their operation and business practices using the methods and practices of Dr. Deming. With pervasive emphasis, the methods of statistical process control and continuous improvement were taught to managers and workers alike. For those of you who are old enough to have experienced that quality training, I am certain you will recall vividly the "Red Bead" experiment, which opened our eyes and minds to the concept of natural variation. If you have never heard of the experiment, I highly recommend doing a bit of research; it will be well worth your time.

Along with the increased focus on quality came Deming's idea of "profound knowledge." Profound knowledge could never be achieved with "job hopping" managers and employees. Dr. Deming espoused that deep understanding of the company and its products only comes from years of experience and progression within the organization. Deming insisted that quality improvement required having complete understanding of the process by which the products of the business were made. Dr. Deming, in his characteristically blunt style, acerbically denigrating management, most likely would have said it this way, "How can you improve if you do not know what you are doing?"

Other extremely notable influences to the quality revolution in the U.S. came from Joseph Juran and Philip Crosby. Juran focused on the education and training of management and the human relations problem of resistance to change. The "Pareto principle,"[2] was introduced to the vocabulary of quality due to the work of Juran. Philip Crosby's book, Quality is Free, made, unequivocally, the business case for quality and the improvements it offered [1]. Succinctly stated, the investment and implementation of a good quality system will pay for itself many times over. Crosby also put forth the Quality Management Maturity Grid, which represents the characteristics of the quality system using five evolutionary stages: (1) uncertainty, (2) awakening, (3) enlightenment, (4) wisdom, and (5) certainty. By utilizing the grid, businesses have a template for understanding and improving their quality system.

Quality Culture

The startling success of Japanese business, coupled to the loss of market share along with project failures in the U.S., created the impetus for dramatic change. The terminology describing this abrupt departure from present business practice and culture is "paradigm shift." These words have become commonplace and are integral to the jargon of those involved in process and quality improvement today.

Out of the desperate desire to improve and the recognition of quality as the pathway came the creation of the SEI in 1984 and the first Project Management Body of Knowledge (PMBOK® Guide)[3] in 1987. To heighten the emphasis for embracing the culture of quality, the U.S. government in 1987 created the national award for performance excellence, the Malcolm Baldridge National Quality Award.[4] The award was intended to incentivize and recognize U.S. businesses for achieving world-class quality. To receive the award a company must show excellence in seven areas of performance: (1) leadership, (2) strategic planning, (3) customer focus, (4) measurement, analysis, and knowledge management, (5) workforce focus, (6) process management, and (7) demonstrable results.

Possibly the most recognized contribution of the SEI to improving the software development process and product quality was the creation of the CMM®. Through Watts Humphrey's initial work [2], the CMM evolved from the adaptation of Crosby's Quality Management Maturity Grid to a staged improvement approach for software development [3]. The CMM is characterized by five levels of process maturity: (1) initial, (2) managed,

(3) defined, (4) quantitatively managed, and (5) optimizing. The CMM provided software organizations a template for improvement that could be objectively assessed. Evidence supports the assertion that software projects performed by organizations attaining maturity levels 4 and 5 are significantly more likely to deliver products that satisfy the requirements of the customer [4]. Although the SEI focused its efforts toward military software, primarily U.S. Air Force systems, the CMM[5] came to be used extensively by commercial software companies, as well.

The PMBOK, now in its fourth edition, is the recognized embodiment of the knowledge and practice of project management [5]. Professional project management is presented as activities for nine knowledge areas[6] occurring over the five life-phases[7] of the project process. The quality improvement view of the Project Management Institute (PMI®) is that by standardizing the methods in the PMBOK and certifying managers through the Project Management Professional (PMP) examination, improvement in project results can be expected. That is, by increasing the number of project managers knowledgeable of the best practices, a growing percentage of projects should complete with good quality, on time and within budget.

Both the SEI and PMI have the same objective of institutionalizing quality in organization, process, and product. However, in comparing the two approaches it is observed that an organization utilizing the PMI method would likely be rated, at best, as maturity level 3 (defined) of the five levels defined for the CMM. The CMM makes a distinction between desirable characteristics for projects and organizations, whereas it is not so clear in the PMBOK. Depending upon how organizations approach using the PMBOK, there may not be company policy for managing its projects. If management methodology is inconsistent and not tailored to the application from the standard for the organization, the best the company could be rated is CMM level 2 (managed).

The more significant difference is the aspiration for each of the two approaches. The CMM seeks continuous improvement, whereas the PMBOK with the PMP certification is limited to the improvement offered by standardization. The CMM approach at level 4 seeks evidence of management's use of data for project control and process improvement. Also, this maturity level requires a quality system that prevents defects from propagating through the process. At level 5, the application of statistical process control is utilized to understand process changes intended to reduce the natural variation in the organization's processes [6]. Achievement of levels 4 and 5 leads to the application and the long term benefits of knowledge management.[8]

The PMBOK mentions the use of data and measures for performance reports and has a brief discussion of Earned Value Management (EVM) as a method for project control.[9] Furthermore, the PMBOK alludes to having and using project performance data and quality measures, but there is little verbiage compelling a project manager or his/her organization to be data driven.[10] Without performance measures and indicators, management decisions come solely from experience and intuition. Does it not make sense for managers to be as well informed as possible concerning their project's performance? And does it not also seem reasonable that better informed decisions increase the probability of a successful project outcome?

Similarly, making systemic improvement has little basis when measures and indicators are not ingrained in the organizational culture. How is it known an improvement is needed? And, after a change is introduced, how can management know if improvement is achieved when there is no or scanty evidence of how the present process performs or of the quality of its products? Likewise, when measurement and analysis is not common practice, there is low need for the application of knowledge management for improving project planning and understanding long term process improvement and performance drift.

Improving the Practice

The message to this point should be obvious: the PMBOK establishes a standard for good practice, but does not promote a culture of continuous improvement. Unlike the CMM, there is no assessment to see if the best practices of the PMBOK are implemented and performed well. Without having an understanding of whether or not best practices are used, how can success or failure of a project be evaluated? How can the organization improve its methods and policy, thereby providing an environment where projects are delivered successfully, waste is reduced, and business flourishes?

The methodology intended to fill this void is the Organizational Project Management Maturity Model (OPM3). The project management model for improvement was issued initially in October 2003 and was later updated in December 2008 to align with the fourth edition of the PMBOK. OPM3 is a best practice standard for assessing and developing project management capability. It is an approach for understanding project management behavior and bringing focus to areas of performance needing improvement.

OPM3 is meant to serve the field of project management in a similar manner to the CMM for software process improvement. The improvement stages ascribed to OPM3 are (1) Standardize, (2) Measure, (3) Control, and (4) Continuously Improve. The process characterization for each of these four stages is very much the same as those for the software model. Initially, the organizational processes are standardized. Once standardization is in place, measurement of the process can proceed. Having measures in place, controlling and subsequently improving the process become possible.

The OPM3 project domain framework identifies nine process areas that show correspondence between PMBOK processes and OPM3 best practices [7]. Of the 44 PMBOK processes within the nine areas, only four directly relate to project execution: schedule control, cost control, quality control, and risk monitoring and control.

From the viewpoint that execution utilizes the most project resources over the longest phase of the project, it would seem appropriate that the methods and tools for these important control processes would be discussed in detail. Although Measure is an important stage in the OPM3 approach to improvement, there is minimal guidance for what constitutes its successful achievement. OPM3 does describe the characteristics of measures, but to progress and advance to the Control and Continuously Improve stages something more specific would be helpful.

The Way Forward

To emphasize the importance of measures, the quotations of Lord Kelvin are often used. One especially makes the point:

"In physical science the first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it. I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be [8]."

Although Lord Kelvin is addressing his comments toward the hard sciences, such as physics and chemistry, his point is equally applicable to project management. When a project manager does not have objective measures of performance for cost and schedule, he/she cannot react intelligently and, consequently, has little chance of guiding the project to successful completion. Under these circumstances, the manager has only his/her personal knowledge and intuition as a basis for action.

As discussed earlier, EVM is mentioned only briefly in the PMBOK as a "Tool and Technique" for controlling cost and schedule performance. Furthermore, OPM3 identifies the performance measures and indicators from EVM as merely an approach to be considered for satisfying the Measure stage of project management improvement. Unquestionably, the power and usefulness of the earned value methodology has not been exploited to the degree it should be. Therefore, it becomes arguable that the lack of emphasis from these two principal documents, regarding EVM, has slowed the advancement of the project management profession to the "state of Science."

When the performance of a project is known in qualitative terms, we can say we know something about it. However, in general, the qualitative description is not enough information for analysis and management action. Only when performance is described by objective measures can project managers truly gain deeper understanding and formulate reasoned tactics for improving the opportunity for success.

EVM is more than 40 years old; a well-defined project management methodology, which has the capability to provide the quantitative measures to advance project management to the level of science. It is supported by standards [9,10], textbooks[11], an improvement model [11], training[12], certifications for both individuals[13], as well as organizations[14], and automation applications are readily available from several vendors[15]. As all of the footnotes associated with the previous sentence attest, EVM is a well-developed technology with considerable infrastructure. EVM, in fact, is approximately 20 years older than the PMBOK and possibly more mature in its application.

The known capability and availability of the management method lead us to the question, "Why is not the use of EVM more prevalent?" The reasons cannot be stated with certainty, but the following is offered as a rational summation for consideration. In its beginnings, EVM was imposed on defense contractors performing development of major weapon systems. In the late 1960s and throughout the 1970s, the creation of custom EVM systems for each application was not a simple matter. The computing capability to connect time accounting, the project schedule, earned value (work accomplished), and actual costs was expensive to develop. EVM was in its infancy, as was the necessary computing technology to make its use practicable. The early EVM systems were very likely cumbersome to use and not that accurate either. All of these things created the prevailing reputation that EVM is terribly complex, difficult to do, overly burdensome to employees and managers, and expensive to create and implement. When this is the perception, the likelihood of employing EVM is very low. It is contended that this attitude persists and is prevalent within the project management community today.

This negative reputation for EVM, however, is not the present circumstance, at all. As expressed earlier, there is considerable support available. EVM can be implemented and applied without undue difficulty. Possibly the most troublesome hurdle to implementation is the reporting of earned value; i.e., assessment of project accomplishment. Disciplined reporting is a difficult transition to make for most, people and organizations, as well. However, once reporting becomes a commonplace expectation, an environment of transparency and accountability is created for everyone involved. Both characteristics are most assuredly desirable outcomes. Certainly there are more implementation hurdles, but generally, these pertain to the need or desire for having a sophisticated, or even a certified EVM system.

Of significant importance is the realization that the elements prescribed by the PMBOK to prepare the project for execution are the necessary ingredients for applying EVM; i.e., Work Breakdown Structure, estimates of task cost and duration, task sequencing, and creation of the schedule. The additional step of aggregating the information into the Performance Measurement Baseline[16] creates the necessary reference for EVM performance analysis. The key point from this discussion is that, when the accepted project management guidance is utilized, taking the next step to employ EVM is not an overwhelming undertaking. Conversely, when employing EVM is the organization's standard method of project control and reporting, it encourages and re-enforces PMBOK guidance and OPM3 best practice. Also, once implemented, EVM greatly facilitates improvement to project management practice, and thereby promotes achievement of the higher levels of OPM3: Measure, Control, and Continuous Improvement.

EVM has a primary focus on the cost aspect of projects, but does have indicators for assessing schedule performance. However, these schedule indicators are limited in usefulness due to their flawed behavior for late performing projects. To overcome this deficiency, Earned Schedule (ES) was created in 2003 [12]. ES extends EVM and provides reliable analysis of the schedule performance.

Together, EVM and ES provide incredible capability for measuring and analyzing project performance. With the employment of EVM project managers can assess present cost performance status, forecast final cost, and determine performance necessary to meet the cost objective. In an analogous manner, the application of ES provides the ability to perform schedule analysis;

i.e., report status, forecast completion, and determine the future performance required to achieve the desired completion date. Additionally, ES introduces a new concept, schedule adherence. The measure of schedule adherence increases understanding of how the project is being performed. The concept yields the ability to analyze critical path performance, identify constraints, impediments, and potential areas of rework. Furthermore, when project performance is poor, ES used with EVM gives project managers the ability to develop tactics for recovery. It should be clear from this discussion that the numerical methods inherent with EVM and ES provide the ingredients to propel project management to the "state of science."

Beyond the application to monitoring and controlling the project in its execution phase, the numerical data contribute to creating a project archive. The execution history, aggregated with other project documents, form a complete project record. The assembly of formalized project records further promotes making the data useful for the planning of new projects and for analysis of improvement initiatives. As a natural consequence, without emphasis, the organization will gravitate to the employment of knowledge management.

Through the use of EVM with ES, the argument is made that project performance will improve as well as the organizational practice. The numerical evidence of performance with the accompanying analysis capability, as a result of their application, provides primary input to the achievement of the higher levels of OPM3. Performance measures are available for stage 2 (Measure). Analysis of the measures and derived indicators yield methods of project control necessary to achieve stage 3 (Control), and the application of knowledge management facilitates the accomplishment of stage 4, Continuous Improvement.

A quantum advance for project management is readily available through the implementation of EVM and its ES extension.

Summary

Quality in the 1980s became the driving force for product and process improvement. The approach for achieving quality is derived from the initial work of Walter Shewhart, with subsequent evolutions contributed by Deming, Juran, and Crosby. Building on the significant work of these men, Humphrey and the SEI formalized the quality system for organizational application to software development. Subsequently, PMI adapted the ideas and concepts from the SEI to project management.

The embodiment of quality for project management is the collection of best practices included in the PMBOK, while the methodology for improvement of the practice is contained in OPM3. The observation is made that EVM and ES are not sufficiently emphasized by the two PMI documents. Implementing EVM and ES is encouraged and shown to reinforce good practice and support quality. The stated expectation from the application of EVM along with ES is improvement in project performance, while advancing and maturing organizational behavior. The proposition is made that the application of the system of measures and analysis methods from EVM and ES advances project management to the "state of science." And ultimately, achieving this state leads to knowledge management and continuous improvement.

Thank You CrossTalk!

This last fall in my conversations with the CrossTalk staff while finalizing an article for the November-December 2012 issue, I was asked if I would be interested in submitting an article for the 25th anniversary of CrossTalk. "Of course!" I replied. I am certain all of the other authors in this historic issue feel as I do …very flattered to have been asked.

Possibly some of the long-time followers of CrossTalk recall my name, but I doubt most of today's readers have any knowledge of me. As a bit of history, I began submitting articles for CrossTalk publication in 1999. From then through 2012, seventeen articles were published. This is my eighteenth.

I have published 45 articles in nine other journals, including an international highly refereed publication. The process CrossTalk uses to first qualify the article and then improve it is by far the most thorough and toughest of any of the journals with which I have experience. I vividly recall many of the telephone conversations concerning reconciling reviewer comments with then publisher, Beth Starrett. She had Bulldog tenacity for getting it right.… As angry as we would sometimes become with each other, the process proved time and again to greatly improve my article.

Over the years Beth, the article coordinator, Nicole Kentta, and I became friends. There were several times during the STSC conferences I would join Nicole, Beth and her family for dinner … wonderful experiences, which I cherish in my memories. Thank you Beth for your friendship and all of the work you did making my writing efforts better.

For this issue, I struggled with what I might submit. I believe my choice is in keeping with the "roots" of CrossTalk; i.e., software process improvement. The topic of my article is consistent with my previous publications and is at the heart of improvement …performance measurement.

Beth said to me many times, "Your article gives me a headache!" My articles generally had mathematics which she did not enjoy. Possibly, Beth will enjoy this article and hopefully you will, as well. It has no mathematics. Nevertheless, I believe its message is important.

-Walt Lipke

Disclaimer:

CMM® and CMMI® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ABOUT THE AUTHOR

Walt Lipke retired in 2005 as deputy chief of the Software Division at Tinker Air Force Base. He has over 35 years of experience in the development, maintenance, and management of software for automated testing of avionics. During his tenure, the division achieved several software process improvement milestones, including the coveted SEI/IEEE award for Software Process Achievement. Mr. Lipke has published several articles and presented at conferences, internationally, on the benefits of software process improvement and the application of earned value management and statistical methods to software projects. He is the creator of the technique Earned Schedule, which extracts schedule information from earned value data. Mr. Lipke is a graduate of the USA DoD course for Program Managers. He is a professional engineer with a master's degree in physics, and is a member of the physics honor society, Sigma Pi Sigma (SPS). Lipke achieved distinguished academic honors with the selection to Phi Kappa Phi (FKF). During 2007 Mr. Lipke received the PMI Metrics Specific Interest Group Scholar Award. Also in 2007, he received the PMI Eric Jenett Award for Project Management Excellence for his leadership role and contribution to project management resulting from his creation of the Earned Schedule method. Mr. Lipke was recently selected for the 2010 Who's Who in the World.

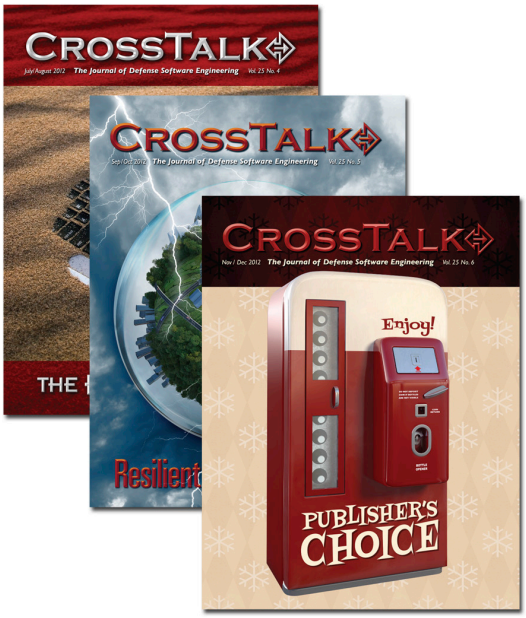
E-mail: waltlipke@cox.net
Phone: 405-364-1594

REFERENCES

1. Crosby, Philip B. Quality is Free, Penguin Books, New York 1979
2. Humphrey, Watts S. Managing the Software Process, Addison-Wesley, New York 1989
3. Paulk, Mark C., Weber, Curtis, Chrissis. The Capability Maturity Model: Guidelines for Improving the Software Process, Addison-Wesley, Boston 1995
4. Goldenson, Dennis R., Gibson, Ferguson. "Evidence About the Benefits of CMMI," SEPG 2004 <http://www.sei.cmu.edu/library/assets/evidence.pdf>
5. ANSI/PMI 99-001-2008, A Guide to the Project Management Body of Knowledge, PMI, Newtown Square, PA 2008
6. Pitt, Hy. SPC for the Rest of Us, Addison-Wesley, Reading, MA 1994
7. Northrop, J. Alan. Every Organization Can Implement OPM3, Triple Constraint Inc., Marion, IA 2007
8. vLord Kelvin quote is from <http://zapatopi.net/kelvin/quotes/>, October 2010
9. Earned Value Management Systems, ANSI/EIA 748-B, Arlington, VA June 2007
10. Practice Standard for Earned Value Management, PMI, Newtown Square, PA 2005
11. Stratton, Ray W. The Earned Value Management Maturity Model, Management Concepts, Vienna, VA 2006
12. Lipke, Walt. "Schedule Is Different," The Measurable News, March 2003, 10-15
13. Lipke, Walter H. Earned Schedule, Lulu Publishing, Raleigh, NC 2009

NOTES

1. This article was originally published in PM World Today online journal (December 2010). The article is no longer accessible as the journal ceased publication with its last issue in March 2012.
2. Pareto principle: eighty percent of the problems come from twenty percent of the causes.
3. For brevity, PMBOK Guide is shortened to PMBOK hereafter.
4. The Malcolm Baldridge Award has its basis in the The Malcolm Baldridge National Quality Improvement Act of 1987.
5. Although the CMM has evolved to the CMMI, only the former is referenced for the purpose of this paper.
6. Knowledge areas: integration, scope, time, cost, quality, human resource, risk, procurement
7. Project phases: initiation, planning, executing, monitoring & controlling, closing
8. Knowledge management is the deliberate effort of an enterprise to gather, organize, refine, and disseminate knowledge, tacit and explicit, concerning its practices, processes and products for the purposes of retention and transference.
9. Reference PMBOK 7.3.2 (Control Costs: Tools and Techniques)
10. Reference PMBOK 4.4.1.2 (Performance Reports), 10.5.3 (Report Performance: Outputs)
11. Several text books are available. One I highly recommend is Project Management Using Earned Value, Humphreys & Associates, Inc., Orange, CA 2002.
12. Very good training is readily available. The following sources are well respected: Humphreys & Associates, Performance Management Associates, and Management Technologies. A good analysis course is available from Project Management Training Institute.
13. For individuals, the certification process to obtain the credential of Earned Value Professionalä is administered by the Association for the Advancement of Cost Engineering International.
14. For organizations, The Defense Contracts Management Agency certifies compliance to the requirements of the ANSI/EIA 748-B standard
15. A few sources for EVM tools are: Deltek, Dekker, Primavera, Artemis, ProTrack, ProjectFlightDeck, EVEngine, and Microsoft Project.
16. The PMB is the time phased budget plan used as the reference for project performance analysis.



CALL FOR ARTICLES

If your experience or research has produced information that could be useful to others, CROSSTALK can get the word out. We are specifically looking for articles on software-related topics to supplement upcoming theme issues. Below is the submittal schedule for three areas of emphasis we are looking for:

Legacy System Software Sustainment
Jan/Feb 2014 Issue
Submission Deadline: Aug 10, 2013

Mitigating Risks of Counterfeit and Tainted Components
Mar/Apr 2014 Issue
Submission Deadline: Oct 10, 2013

Please follow the Author Guidelines for CROSSTALK, available on the Internet at <www.crosstalkonline.org/submission-guidelines>. We accept article submissions on software-related topics at any time, along with Letters to the Editor and BackTalk. To see a list of themes for upcoming issues or to learn more about the types of articles we're looking for visit <www.crosstalkonline.org/theme-calendar>.

A New Software Metric to Complement Function Points

The Software Non-functional Assessment Process (SNAP)

Charley Tichenor

Abstract. Sizing software requirements is an essential best practice in software project management for forecasting the work effort required for software development projects (and other related metrics). Arguably, the currently most accurate software metric for measuring the size of software is the International Function Point Users Group (IFPUG) "function point," which has the ISO standard ISO/IEC 20926:2009. Function points basically measure the size of the data flow and storage through the software, which we define in this paper as "functional" requirements. But function points do not measure other software requirements, which also require work effort resources. IFPUG has recently completed a successful beta test of a new method to assess the size of other, "nonfunctional" requirements, which when used in conjunction with function points should further increase the accuracy of software forecasting. The authors believe that this Software Non-functional Assessment Process v. 2.0 (SNAP) is ready to enter industry and academia for initial practice and further research.

Introduction

Forecasting the cost to produce software has been transformed from an art into largely a science through a methodology called function point analysis. Function point analysis basically quantifies the volume of data flow and storage through the software application; based on this measurement the cost required to develop the software can be quantitatively forecast. Years of experience with function points has shown it to be a robust methodology [1]. Yet, one wonders if a complementary software metric could be developed and used along with function points so that data flow and storage, and other aspects of the software that function points do not consider can be measured. Combining these measurements should improve the quality of software development cost forecasting (and other software metrics).

One proposed complementary metric is from SNAP. IFPUG, through its Non-functional Sizing Standards Committee, SNAP Project Team, developed a procedure for SNAP and wrote the SNAP "Assessment Practices Manual," now in version 2.1 [2]. During August and September 2012, the SNAP team conducted a beta test to measure how well SNAP 2.0 correlated with work effort. This beta test was successful, and the purpose of this paper is to share the results of this beta test. We will discuss:

a. **What function points are**
b. **What SNAP is**
c. **Why SNAP may be important**
d. **How the beta test was conducted**
e. **What the results were**
f. **Areas for future research**

Review of the Related Literature

IFPUG is the largest software metric association in the world, with more than 1,000 members and affiliates in 24 countries. The non-profit International Software Benchmark Standards Group (ISBSG) has become the largest source of benchmark data, with more than 5,000 projects available. New benchmarks are being added at a rate of perhaps 500 projects per year. All of the ISBSG data is based on function point metrics [3].

IFPUG maintains arguably the most widely used functional software sizing metric in the world, the IFPUG "function point" (in this paper, we will always refer to the unadjusted function point). The IFPUG Counting Practices Manual [4] is one standard for measuring functional requirements, and is recognized by the ISO.

ISO/IEC 20926:2009 specifies the set of definitions, rules and steps for applying the IFPUG Functional Size Measurement method. ISO/IEC 20926:2009 is conformant with all mandatory provisions of ISO/IEC 14143-1:2007. It can be applied to all functional domains and is fully convertible to prior editions of IFPUG sizing methods. … ISO/IEC 20926:2009 can be applied by anyone requiring a measurement of functional size. Persons experienced with the method will find ISO/IEC 20926:2009 to be a useful reference [5].

A function point is like a "chunk" of software. It is similar in concept to a "square foot" of house size, a "kilometer" of distance, a "gallon" of gasoline, or a "degree Kelvin" of temperature. According to IFPUG's Counting Practices Manual, function points are assigned to different components of software according to the user's viewpoint (rather than the programmer's viewpoint). IFPUG recognizes five different types of software components, listed in the table below, that are basically measures of the data flow and storage through the software. Also listed are their relative sizes in terms of function points and based on their complexity levels.

	Low	Average	High
External Input	3	4	6
External Output	4	5	7
External Inquiry	3	4	6
Internal Logical File	7	10	15
External Interface File	5	7	10

Table 1

For example, an input screen process for entering data into an application might be measured as a low complexity external input worth three function points, and a high complexity external interface file is counted as 10 function points. The IFPUG Counting Practices Manual has repeatable standards for how to count function points and determining whether a component has low, average, or high complexity.

Data Operations	Technical Environment
Data entry validations	Multiple platforms
Extensive logical and mathematical operations	Database technology
Data formatting	Batch process
Internal data movement	
Delivering added value to users by data configuration	
Interface Design	**Architecture**
User interface methods	Mission critical/real time systems
Help methods	Component based software
Multiple input methods	Multiple input/output interfaces
Multiple output methods	

Table 2

Here is how we can use function points for forecasting the cost to develop software. First, as an analogy, suppose that a customer wants to build a new house in a certain community. Suppose further that a typical house in that community is built at a cost averaging $300 per square foot. If the customer wants a new house of 1,000 square feet, then a good estimate of its cost will be about $300,000. Suppose we are considering building a new software application. Before we start building it we want to forecast its cost. A qualified function point analyst starts by examining the software's data requirements. Then, using the standards in the IFPUG Counting Practices Manual, the analyst counts each instance of the components in Table 1 that are anticipated to be in the software, and then totals their values for the final function point count. (adapted from [6]).

This function point size correlates with development cost. The original paper showing that function point size correlates with development cost was published in 1977 by Dr. Allan Albrecht in his paper "Measuring Application Development Productivity [7]." This paper was the publication of the results of his research team's development of the initial version of the function point methodology at IBM. The team correlated function point size of various IBM applications with their corresponding work effort, and found the correlation to be statistically significant. Since the publication of this paper, numerous organizations have developed function point-based software productivity models to help them forecast software development costs. Some companies have compiled large amounts of such data from government, industry, and other sources, and built commercial software estimation tools which use function points and other productivity indicators (such as software language used, skill of the programming team, project management tools used, etc.) to help clients forecast their software development costs.

Now we can forecast the cost to develop this software. Suppose that the function point analyst identified the software's components from Table 1 and counted a total of 1,000 function points. Suppose further that a typical application of this type is built at a cost averaging $300 per function point. A good estimate of its total development cost is therefore about $300,000.

A reading of the IFPUG Counting Practices Manual indicates that function points are basically a measure of the size of the data flow and storage through the software. For this paper, we define these software requirements as "functional" requirements. The cost estimate of $300,000 for developing 1,000 function points of software is based on data flow and storage size—the functional requirements for the software.

Let us return to our house cost forecasting analogy. A new house of 1,000 square feet in size in this Community should typically cost about $300,000, but the particular house design this customer wants is a little different than "typical." Suppose that this customer also wants to add hardwood floors (instead of typically carpeted floors), a wood-burning fireplace, a refrigerator with an extra large freezer, and extensive wiring to support a special home entertainment system. We improve the cost estimate for this house by factoring in the additional costs of these extras.

Now, suppose we want our software cost estimate to factor in software requirements which are not included as functional requirements in the IFPUG Counting Practices Manual. Let us consider certain requirements within the following categories and their subcategories. These are from the SNAP Assessment Practices Manual (refer to Table 2).

In this paper, we define these kinds of software requirements as "non-functional" requirements because they are not included in the ISO standard function point methodology in the IFPUG Counting Practices Manual yet require additional work effort to develop. We want to assess the size of these non-functional requirements for applications. We also want to know if non-functional size statistically correlates to the corresponding work effort—like function points do. This was the fundamental paradigm of the SNAP beta test.

We want to base the beta test analytics on statistical methods. We include the notions of random sampling, regression models, the F test, p-values, the Runs test, and the Spearman test. Basic Statistics books (for example, [8]) treat these. The next paragraphs will discuss the intended testing analytics.

For the beta test, random sampling means that we collect SNAP sizes from a wide variety of applications across the world. As much as possible with the resources we have, we want to have a sample that represents the software development industry.

Regression is a way to find the correlation between two variables. In this beta test, we want to determine if there is correlation between the SNAP sizes of the applications and their corresponding work efforts. We believe that as the SNAP size increases, the work effort to build those SNAP sizes should also steadily increase.

Statisticians often look for several indicators to measure the degree of strength of the relationship within a set of two variables, in this case, the SNAP size and corresponding work effort. If there is causation, then one indicator (in this case) would be the degree to which SNAP size accounts for the amount of

work effort. This is measured by the r^2 statistic. For example (assuming causation), if our data's r^2 is measured to be .75, then we conclude that SNAP size accounts for 75% of the reason for the work effort.

Another statistic is the associated p-value for this, also called "Significance F" in Excel. The p-value is the probability that we are wrong in concluding that SNAP size is correlated to work effort. If the p-value is .05, then we are 5% sure that we are wrong in concluding such a correlation, or put another way, we are 95% sure that we have statistical significance.

There are some technical assumptions in the standard regression process. One is that the data points are randomly scattered about the regression line. We can test for this using the Runs test, and we are comfortable that the model passes the Runs test if its p-value is below .05.

We also want to test for correlation using the Spearman test. This is a nonparametric test for rank correlation and makes no technical assumptions about the distribution of the data, other than it is randomly scattered about the regression line. This is a "worst case scenario" test we use should we have doubts about the validity of the standard regression test.

The final statistical test is for compliance with Benford's Law. Benford's Law is an interesting statistical test. Software development is a human stimulus and response activity. Part of the overall stimulus for developing software is the need for the non-functional requirements. The response is the number of SNAP points generated. If this occurs, then we can look at the leading digits of the SNAP size. For example, if the SNAP size is 483, then we would consider the leading digit of "4." Benford's Law says that in these stimulus and response situations, the distribution of the leading digits is logarithmic, as in the table below, i.e., 30.1% of the SNAP sizes should start with the number "1," 17.6% of the sizes should start with "2," and so forth until we should measure "9" as the leading digit in about 4.6% of the SNAP sizes [9].

First Digit	Percentage of Occurrences
1	31.10%
2	17.60%
3	12.50%
4	9.70%
5	7.90%
6	6.70%
7	5.80%
8	5.10%
9	4.60%

Table 3

This compliance with Benford's Law happens with function points. A study presented at the 2009 Fourth International Software Measurement & Analysis conference [10] showed that for a large internationally collected sample of function point counts (more than 3,000 function point counts from ISBSG, Victoria, Australia), their leading digits followed the distribution predicted by Benford's Law almost exactly.

Although the SNAP sample will be much smaller, we hope to see good convergence towards Benford's Law.

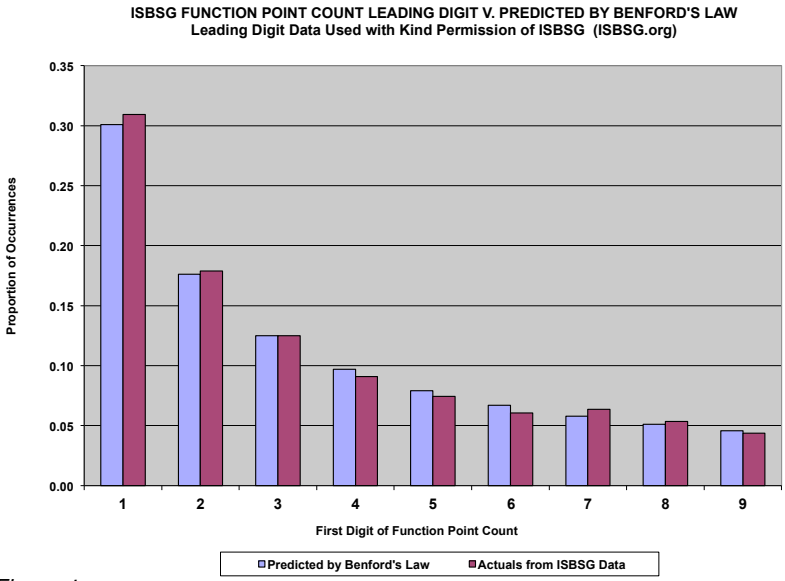


Figure 1

Research Design and Methodology

The purpose of this beta test was to repeat and extend the spirit of Dr. Allan Albrecht's statistical analysis of the early function point methodology for the SNAP methodology. Dr. Albrecht's research showed that software size measured in function points correlated with work effort for the applications tested. In a similar manner, based on data collected from the beta test, our research will hopefully determine the degree to which SNAP sizes correlate with corresponding work effort. Here is our research design and methodology.

Use version 2.0 of the SNAP manual as the basic reference.

Develop a standard SNAP data collection spreadsheet, largely based on last year's spreadsheet. This new spreadsheet had four worksheets:

1. "Basic Instructions" worksheet, which provides detailed instructions for data collection for the SNAP counter.
2. "Application Data" worksheet, for entering descriptive data.
3. "SNAP Counting Sheet," for entering the SNAP points. This worksheet permits the SNAP counter to enter only basic data per SNAP item, such as "DETs," "FTRs," "person-hours," and other data described by the SNAP training. The worksheet then automatically calculates SNAP points. All calculation cells are locked.
4. "Recap" worksheet, which automatically totals the SNAP sizes and work effort.

Issue a call for volunteer SNAP counters, and train them. This training will be done both using written materials (primarily the SNAP Assessment Practices Manual) and by telephone. The counters will choose the applications to size. Hopefully, this call for volunteers will result in a wide variety of countries represented and application types chosen.

Conduct all SNAP sizing at the application boundary level—"application boundary" as defined in the IFPUG Counting Practices Manual.

Collect at least 30 applications' worth of SNAP sizes with corresponding work effort in person-hours. This is to hopefully ensure a statistically large sample size.

If corresponding function point and work effort data can also be collected, then so much the better. This permits additional research. However, such function point counting data is considered optional.

Collect application descriptive data such as types of applications, types of industry, types of software, etc. This data may be used to help improve correlations. However, maintain source confidentiality.

Conduct the beta test throughout August and early September 2012. During the beta test, after counters finish with individual application SNAP sizings, they are to email their data collection spreadsheets to IFPUG. These data sheets will be then "cleaned" of any source information to maintain confidentiality, and then will be forwarded to one of several members of the SNAP team who will perform a "quality control" of the data collection.

As the SNAP data pass "quality control," they will be then forwarded on for statistical analysis.

The beta test analytics will consist of trying to determine the degree of statistical significance using the following tests. First, we will test the data plotting the SNAP sizes of the applications on the x-axis as the independent variables, and the effort expended on the y-axis as the dependent variables. We will use simple linear regression, and especially look at the r^2, what Excel calls "Significance F" (which is the p-value of the corresponding F test), and the p-values of the coefficients of the regression line. We will check for the appropriateness of testing for regression using regression through the origin. We will conduct the Runs test and Spearman test, and also test for convergence to Benford's Law. We will also experiment with changing weighting factors and other aspects of SNAP to try to both improve correlation and its degree of realism.

Presentation and Analysis of Data

We collected data from a wide variety of applications. This ensured that the sample was as close to random as reasonably possible. We had SNAP sizes for 58 applications usable for the part of the test correlating SNAP sizes with work effort, and an additional 14 SNAP sizes usable for the Benford's Law test (but did not have work effort data).

Data was collected from the following countries: Brazil, China, France, India, Italy, Mexico, Poland, Spain, UK, and the USA. We collected data from the following industries: Aerospace, Automotive, Banking, Government, Fast Moving Consumer Goods, Financial Services, Insurance, Manufacturing, Systems Integrators and Consulting, Telecommunication, and Utilities.

After reviewing the data, 58 data points (representing 58 software applications) had sufficient SNAP size and work effort data for further analysis. The first statistical test was a simple linear regression analysis for 58 applications with the SNAP sizes on the x-axis, and the corresponding work efforts in person-hours on the y-axis. The graph below shows the results of this regression. NOTE: the actual work effort hours are not shown on the y-axis of the forthcoming graphs; we do not want to imply that the productivity rate found in this beta test should necessarily be used as a benchmark—we feel that this is premature at this point.

The r^2 for this analysis is .33, which basically means that 33% of the reason for the work effort was due to the SNAP size.

A closer analysis of the graph (and Excel regression tables) shows that the trendline crosses the effort axis at about 100

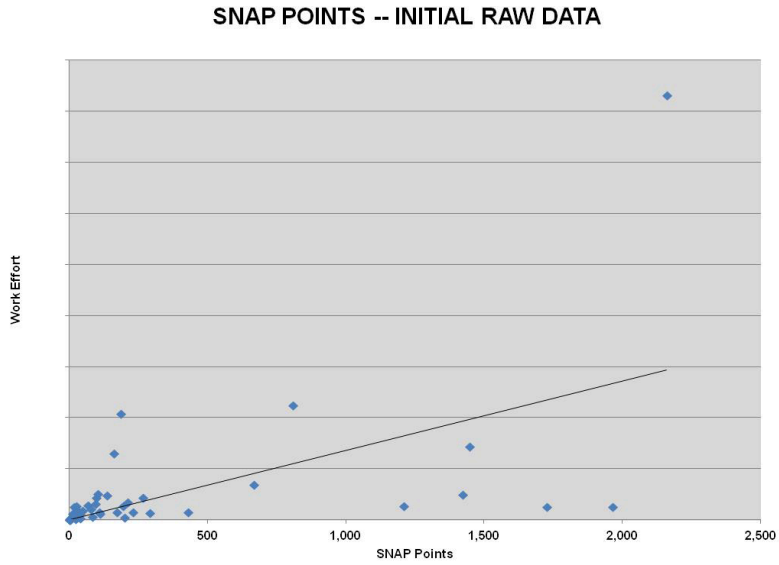


Figure 2

person hours. In theory, this means that if there were zero SNAP points, then the corresponding work effort should be about 100 person hours. This is not reasonable—if there are zero SNAP points then the work effort should also be zero. Therefore, we upgrade the analysis and use a standard technique called "regression through the origin." This forces the trendline through (0,0). This improves the common sense test and increases the r^2 to .41.

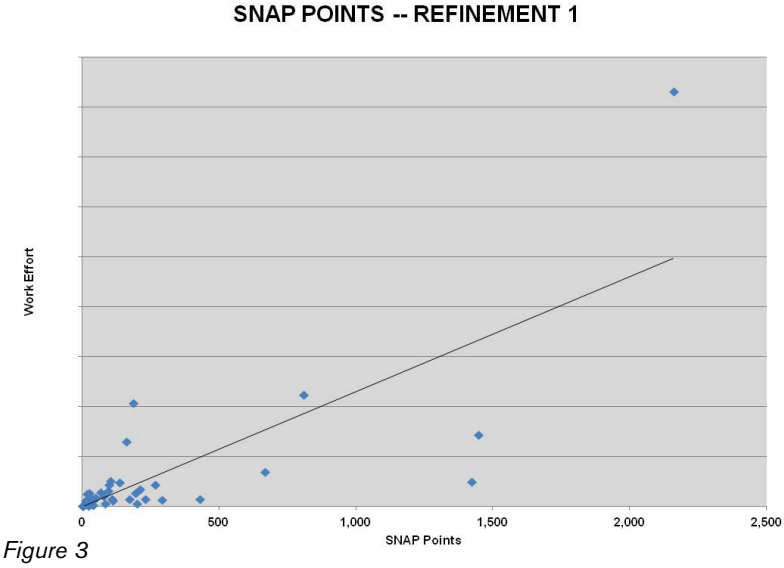


Figure 3

In reviewing the raw data, three applications contained large quantities of Help features. These applications had productivity rates, according to the current version of the model, that were roughly 10 times higher than the other 55 applications. This led us to believe that we may need to reformulate the Help Methods (subcategory 2.2) portion of the SNAP manual. This is an area for future research, so we removed these three applications from the data set. This improved the r^2 from .41 to .66. We later removed seven other applications that counted some Help features, to maintain consistency.

Also, we changed the weighing factors for subcategory 1.5 "Delivering Value Added to Users through Data Configuration"

by changing the weights for low, average, and high from 3-4-6 to 6-8-12. This improved the model's r^2 to .89, with a corresponding Significance F of $1.7 * 10^{-23}$.

To test the requirement that the data points in this model must be randomly scattered about the regression line, we conducted the Runs test. There were 19 runs in the data, which compares favorably with the theoretically optimal 19.96 runs.

We ran the Spearman test for rank correlation. This test produced a rank correlation of .85, with an associated confidence of statistical significance of greater than 99% (p-value <.0001).

The final results of this analysis are on the following viewgraph (refer to Figure 4).

We tested the final version of the results for compliance with Benford's Law. In terms of software development, Benford's Law says that the leading digits in a large portfolio of SNAP sizes should be distributed as in Table 3, repeated below. For example, in a large number of SNAP sizes, about 30.10% of the SNAP sizes should have a leading digit of "1," such as sizes of 15, 139, or 1,728.

First Digit	Percentage of Occurrences
1	31.10%
2	17.60%
3	12.50%
4	9.70%
5	7.90%
6	6.70%
7	5.80%
8	5.10%
9	4.60%

Table 3

Figure 5 shows the SNAP leading digit distribution from the beta test. We used 65 SNAP sizes for this analysis. In general, Benford's Law seems to converge rather slowly, i.e., it requires a very large sample size to "pure out." This SNAP sample size is much smaller than the ISBSG sample size, so the degree of compliance is markedly less; however, we appear to be converging nicely.

Conclusions

We believe that the SNAP Assessment Practices Manual 2.0 has passed the beta test.

a. The test was based on very good sampling techniques
b. The data points are randomly scattered about the regression line, as shown by the Runs test
c. The regression r^2 for 48 projects was .89
d. The Spearman test correlation was .85
e. We are over 99% sure that both tests are statistically significant
f. The distribution of the first digits of 65 SNAP sizes is converging nicely towards Benford's Law

We recommend that the SNAP procedure (with the exception of Help Methods subcategory 2.2) is ready for use by the industry, and is ready for further research.

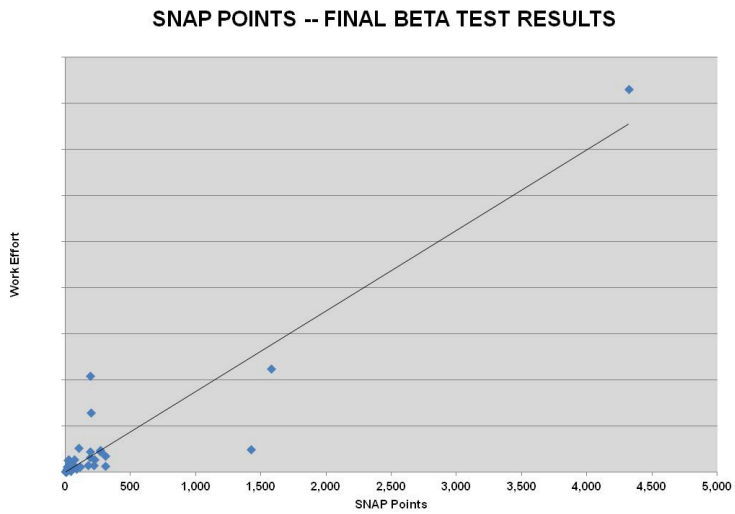


Figure 4:
n = 48 r^2 = .89 Significance F = $1.7 * 10^{-23}$ Spearman = .85 Runs = pass

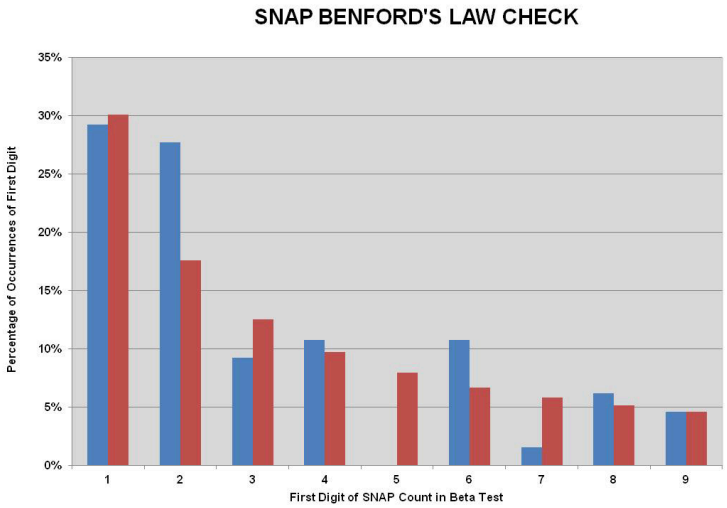


Figure 5

IFPUG has formed a Non-functional Sizing Standards Committee, similar to the Functional Sizing Standards Committee. This committee will continue to develop the SNAP process, encourage SNAP research, develop SNAP training, and maintain the SNAP Assessment Practices Manual.

Areas For Future Research

One possible source of data collection error during the beta test was the experience of the SNAP counters. This was their first use of the SNAP Assessment Practices Manual 2.0. Consistency has been tested for function point counters with very favorable results. Repeat similar consistency tests for SNAP counters after there is much SNAP counting experience in the field.

Continue to experiment with reasonably varying the values of the factors for each subcategory's low, average, and high complexity weights to improve the correlation between SNAP sizes and work effort.

Continue to research the Help Methods, subcategory 2.2.

After a statistically large number of applications have been counted for both function points and SNAP points, conduct research to determine if function points and SNAP points can be combined into a single metric, which correlates to the combined work effort to develop both. Try to combine them like real numbers can be combined with imaginary numbers to produce the complex numbers; try other ideas.

Using a large sample from the ISBSG database, function point counts were tested for compliance with Benford's Law. This almost perfect compliance gave good statistical indication for the soundness of the underlying mathematical structure of function points. After completing a larger number of SNAP sizings (probably over 100), continue repeating this research by testing SNAP sizes for compliance with Benford's Law.

Comments:

This paper is written on behalf of the IFPUG SNAP team. The team developed the SNAP process and published the 130 page "Software Non-functional Assessment Process (SNAP) Assessment Practices Manual," now in version 2.1. The team conducted the version 2.0 beta test to include its research design, the call for SNAP assessors, their training, and analysis of the test results. The team also developed a two-day workshop to introduce the Assessment Practices Manual at the seventh International Software Measurement & Analysis conference in Phoenix, AZ in October 2012.

The SNAP Project Manager and IFPUG Board Member is Christine Green. The IFPUG Non-functional Sizing Standards Committee Chair is Talmon Ben-Cnaan. Other SNAP team members were Wendy Bloomfield, Steve Chizar, Peter R. Hill, Kathy Lamoureaux, Abinash Sahoo, Joanna Soles, Roopali Thapar, Luc Vangrunderbeeck, Jalaja Venkat, and Charlene Zhao.

ABOUT THE AUTHOR

Charley Tichenor is the newest member of the SNAP team, joining in the Fall of 2011 and serving primarily as the team's Statistician. He has been a member of IFPUG since 1991, and was certified as a Certified Function Point Specialist in 1994 and 1997. He has a Bachelor of Science Degree in Business Administration from the Ohio State University, a Master of Business Administration degree from Virginia Tech, and a Ph.D. in Business from Berne University.

Phone: 703-901-3033
E-mail: charles.tichenor@dsca.mil

REFERENCES

1. Jones, Capers, "Sizing Up Software," Scientific American, a division of Nature America, Inc., December 1998.
2. International Function Point Users Group (IFPUG), Software Non-functional Assessment Process Manual, (now in version 2.1), Princeton Junction, New Jersey, USA 08550, 2012.
3. Jones, Capers, "Software Sizing During Requirements Analysis," Modern Analyst, retrieved November 5, 2012 from <http://www.modernanalyst.com/Resources/Articles/tabid/115/articleType/ArticleView/articleId/512/Software-Sizing-During-Requirements-Analysis.aspx>, copyright 2008 by Capers Jones & Associates LLC; all rights reserved.
4. International Function Point Users Group (IFPUG), Counting Practices Manual (now in version 4.3), Princeton Junction, New Jersey, USA 08550, 2009.
5. ISO. "ISO/IEC 20926:2009 Software and Systems Engineering -- Software Measurement -- IFPUG Functional Size Measurement Method 2009," retrieved November 5, 2012 from <http://www.iso.org/iso/fr/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51717>.
6. Dekkers, Carol, "Musings About Software Development," retrieved November 5, 2012 from <http://caroldekkers.blogspot.com/>, 2008.
7. Albrecht, Allan, "Measuring Application Development Productivity," IBM, 1977.
8. Walpole, R. E., & Myers, R., Probability and Statistics for Engineers and Scientists Third Edition, New York, New York, Macmillan Publishing Company, a division of Macmillan, Inc., 1985.
9. Davis, Bobby, & Tichenor, Charley, "The Applicability of Benford's Law to the Buying Behavior of Foreign Military Sales Customers," Global Journal of Business Research, The Institute for Business and Finance Research, (volume 2, 2008).
10. Tichenor, Charley, "Why Function Point Counts Comply with Benford's Law," presented at the Fourth International Software Measurement & Analysis conference, Chicago, IL, 2009.

Improving Affordability

Separating Research from Development and from Design in Complex Programs

Bohdan W. Oppenheim, Loyola Marymount University

Abstract. This paper presents arguments for why defense programs creating physical systems should clearly separate three developmental phases from each other: research, development and design. Research is to be performed first by small teams of scientists addressing the "unknown unknowns" and maturing fundamental science from TRL of 1 to about 3. Next, development of physical modules is to be performed by small and highly specialized engineers. Finally, the system-level design should focus on efficient trading off the module locations, sizes and shapes versus system performance, mass, power requirements, etc. The design with all modules mature and available is equivalent to a car design: to be performed by competent engineers but quite well established. A small cohesive and co-located Program Management team with excellent Systems Engineers and Architects, led by a permanent Program Manager/ Chief Engineer should manage all program phases, assuring smooth transitions between the expert teams and phases. The small weight penalty which may result from the above approach is compensated by orders of magnitude larger savings due to shorter program schedule and optimized engineering effort. Examples are cited.

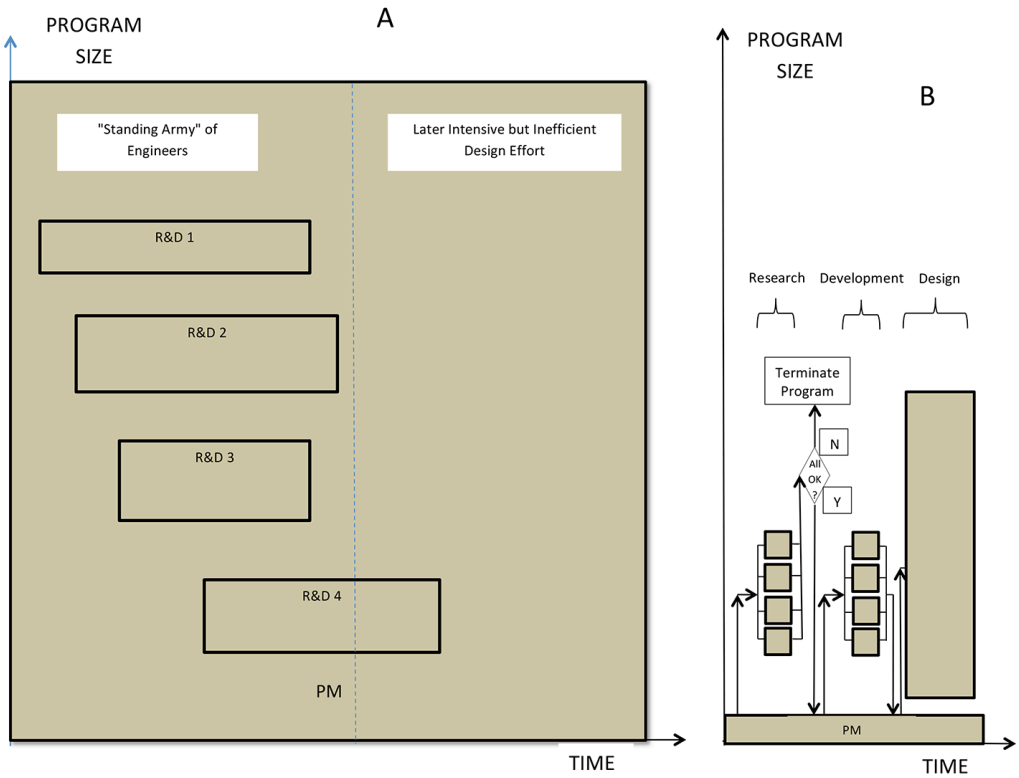
1. Introduction

This paper addresses one of the most fundamental aspects of waste in many large defense programs creating physical systems: the massive waste of engineering labor and time.

It is useful to first review how efficient modern car development is. A typical new car program rigidly adheres to the following phases: 1) First develop all needed components and subsystems (engines, gearboxes, radios, seats, etc.) based on the latest competitive technology and marketing need, and test and validate them thoroughly, preferably in several combinations of sizes, shapes and features, to the level of maturity such that they will be ready for use in new cars. Once all modules are ready to be integrated, and only then: 2) Perform the car design which is a relatively routine problem of trading off the physical module locations, sizes and shapes to fit the styling envelope, performance requirements, vehicle mass and size, powering tradeoffs, etc. Such a design effort has no unknown unknowns, thus no big risks. While requiring towering engineering competence and experience, it remains a fundamentally engineering design: trade-offs and selections of parameters within finite trade space until all requirements are satisfied and some desired performance optimum is reached. Using this approach Toyota completed the Prius car design with new hybrid modules, in nine months from the end of styling to the beginning of error free production—a feat unmatched by any competitor, faster by a factor of 2 to 3 then the next best in class [1].

In contrast, many large complex defense programs in the last decades are contracted "for the entire job" including concept development; co-mingled research, development and design; starting with numerous low-Technology Readiness level (TRL)[1] items. This is usually driven by the perception that cutting-edge technology is more appealing to stakeholders; and the rarely-justified hope that system and technology development can be accomplished in parallel [2].

Starting a large program with very low TRLs and then pursuing research, development and design mixed together under a massive contract is a major source of waste, if not the only one [3]. In effect we pay for a standing army of expensive engineers trying to look busy while small groups of "developmental" engineers frantically try to mature the TRLs. This is illustrated symbolically in Figure 1 A, with the large shaded box symbolizing the entire program effort (or cost) and the four boxes inside it denoting the various inefficient R&D efforts. It is only after the R&D tasks are completed, that the design increases in intensity. A number of aerospace programs notorious for terrible performance followed this pattern, starting with minimal TRL's of one or two, e.g., NPOES [4] and JSF [2]. Numerous other examples are available on the Government Accountability Office webpages.



Typical Program (A) versus Proposed Program (B)

What is worse, the work on the low TRLs is often performed by engineers rather than scientists, using brute-force approach of endless and costly iterations rather than elegant and advanced science methods of rapid trade space exploration and set based design (see Section 2). Starting a large system development with low TRLs causes excessive schedules lasting 10-15-20 years—several times too long when compared to equivalent commercial programs, and costing tens or hundreds of billions of dollars—an order of magnitude too much. The real victim is the war fighter who cannot use the system when needed. In addition to huge original budgets, many programs suffer from major cost growth, and some have to be terminated. Overall, the total cost growth of recent poorly performing defense programs was $295 billion. This practice is in violation of the intent of the Defense Acquisition Logistics: [5] which clearly states that system design should start only after Milestone B, that is after all needed TRLs are quite mature and ready for integration. And this is precisely how commercial companies handle the development at a small fraction of the average defense program cost.

The history of defense and NASA programs offers plenty of examples of successful programs that reinforce the proposed approach, as follows. The Manhattan project which was one of the most difficult programs in human civilization had an efficient research phase during which mathematicians performing hand calculations (before computers!) proved that the nuclear chain reaction would not burn the earth's atmosphere. Once the research was completed, the weapon development and design were completed in weeks. The nuclear submarine project [6] started not with the submarine design but with research on compact nuclear reactors. Once solved efficiently, the development of the nuclear plant and the submarine vessel proceeded predictably and efficiently. The early U.S. space program demonstrated similar advantages [7]. Iridium, one of the technically most successful space programs, is forever a prominent example of technical (if not marketing) efficiency [8].

This article submits that we can adopt a lot of commercial development practices to aerospace programs without sacrificing anything of value, and vastly reduce program schedule and cost, bringing weapons to the war fighter faster and more affordable. The recommended good-sense process is described in Section 2. In Section 3 we discuss the desired management of the entire program, and in Section 4 we identify potential weaknesses and strengths of the approach in the defense environment.

The present approach has been based on several Lean Enablers described in [9, 10] and also listed on the web [11].

2. Ideal Sequence: Research-Development-Design

Occasionally, a set of common words evolve into an idiom which, with frequent use, becomes a paradigm and can be very difficult to eliminate. The words "research and development" seem to be an inseparable pair in this category. This may have been justified in earlier decades of simpler systems. Now when the system complexity is vastly higher, and the research phase needs a dedicated scientific approach, the term has become destructive, costing billions of dollars in inefficient programs. Our task is to clearly untangle three development phases from each other: research using fundamental science, engineering development of modules, and engineering design, as follows (see Fig. 1 B):

The role of research teams is to develop each immature technology to 3 from TRL of 0-1, ending with a demonstration of technical feasibility and validation of the technology. This work phase is driven by global competition: "we need to develop better products, with better technologies all the time". If the technology is challenging, involving significant unknown unknowns, a cost-plus contract may be justified for this phase. But it is critical that the work be done by a very small team (a few individuals is usually sufficient) of highly competent researchers with doctorates in sciences, the love of learning from scholarly journals, and the inner drive to succeed. Each small team should be contracted independently of others, because their areas of expertise do not usually overlap. These folks are rarely engineers. Aerospace design engineers are not needed on these teams therefore large defense programs cannot be justified for this phase; in fact such programs are the opposite of what is needed here. The teams should be protected from defense bureaucracy that would only slow the progress. Even though this phase may be open-ended and contracted cost-plus, the small size of the team(s) assures a modest budget and good progress. Modern science offers a rich body of knowledge on how to make such open-ended challenges efficient and even predictable, using set-based studies [12] trade space exploration [13], and optimized iterations [14]. Since the expenditures are small, a vast bureaucratic oversight should not be needed. If the teams are properly selected for their towering scientific competence, and not sabotaged by bureaucracy, rapid progress can be achieved in schedules lasting from months to a few years. For example, the research phase of the Manhattan project, one of the most difficult programs ever undertaken, took only one year [15].

Development. For each module under development, if and only if the research phase is successful (having achieved TRL of at least 3), a new contract should then be issued to a small focused team of developmental engineers. These engineers are different from scientists and from design engineers and must not be confused with them. The task for a team of developmental engineers is to mature the given TRL from 3 to the mature validated module of hardware, software or a combination, ready to be integrated into a later design. Since this phase has no unknown unknowns, there is no justification for any cost plus work, and the work should be predictable and plannable, with a fixed price and reasonable schedule. Ideally, each module should be packaged into several shape and size combinations, to make subsequent design(s) easier and to promote reusability. The added cost of multiple packaging is a small fraction of the module development effort but has big payoff due to module reusability. The software should also be created with long-term general reusability in mind. This phase calls for solid skills and specialized experience in designing the given module(s). An expert in physical system design may be needed on each team to formulate requirements for the module, which would be consistent with subsequent system design. The requirements should address environmental constraints, use scenarios, top-level interfaces with other subsystems/modules, top-level tradeoffs, and best applicable standards.

System Design. At this time all mature modules should be available for integration. The remaining system design phase involves "routine" tradeoffs between system performance, mass, strength, size, shape, power, years in service, reliability, etc. This is where we need a broad spectrum of system-level engineers and a "systems engineering factory". The system design engineers should efficiently tradeoff the above parameters and select and move the modules around until all constraints are satisfied. This work, even though calling for a high caliber engineering competence, is fairly standard; this is what system design engineers do for a living. There should be no unknown unknowns left at this phase. All high-level technical risks should have been handled in the prior research or development phases. As such, a system-level contract must be contracted as fixed price and reasonably priced and scheduled, based more on commercial program estimates than the bloated defense programs of recent years. Any bidding company who says that they cannot bid a reasonable price in this situation, when all modules are already available, and the top-level requirements are stable should be excluded from consideration for incompetence.

Practically all carmakers follow the described research-development-design sequence, with the best in class demonstrating an amazing overall efficiency.

3. Systems Engineering and Architecting, and Program Management

Ideally, the three phases: research, development, and design should be contracted separately, each to the most qualified teams available for the given phase. Yet, there must be an overall management of the program from the beginning to the end. The following approach is recommended, following [9, 10].

From the program inception, there should be a single and small integrated program management team performing technical management (concept development and systems engineering and architecting), as well as business management (project management, risk management, acquisition, contract monitoring, program monitoring, and supporting functions). This should be a small cohesive co-located team handling the entire program from concept development to Milestone A. Next, the management team should contract and manage first the research phase, then the development phase to Milestone B, followed by the design phase and system integration to Milestone C, including system level verification and validation. The program management should also continue into the operational program phases of transition, operations and logistics, and disposal. This management team should be characterized by the following:

• Co-located minimum-size team. All people should be highly experienced in the system domain. The team must have total responsibility, authority, and accountability for both technical and business success of the entire program.

• The contract should call for managing the entire program during the entire lifecycle.

• There must be a single leader (called "Program Manager" or "Chief Engineer") who is not subject to military rotations, who is the person dedicated to unconditional program success, and who has personal stake in the success (accountability for failure and high reward for success). This excellent leader should be competent in program management, systems engineering, domain engineering.

• Effective team approach: single, co-located, cohesive, and well-integrated team[2].

The management team should manage the following phases of the program:

1. Capture stable system-level customer-need requirements and scenarios of operations (the fewer requirements the better). If these top-level requirements are not stable the program must not be allowed to proceed under any circumstances as this will guarantee budget raptures and risk total failure.

2. Perform enough concept development and system architecting to identify all low TRL (high risk) items, and the overall concept configuration. One year is regarded as plenty of time for a competent team to perform a comprehensive concept development and architecting in response to stable and wise top-level requirements. Modern approaches such as Model Based Systems Engineering [16], or Vienna Development Method [17] may be used in this phase, although the actual approach should be left up to the team and the contract should not be too prescriptive; otherwise it may slow the progress and introduce unnecessary bureaucracy.

3. Research contracts: Then, for each low TRL item, issue a Request For Proposal (RFP) and source select a small team, paying attention primarily to the past scholarly successes and credibility of the teams (illustrated in Fig. 1 B symbolically by four small "research boxes" denoting, say, four needed research topics). All such small contracts for maturation of TRLs should be issued in parallel. Large defense contractors are badly suited for this phase as they tend to activate a large "standing army"—precisely what we are trying to avoid. Monitor all projects in this research phase and wait until all low TRLs reach the level of at least 3. If even one research team fails to achieve success do not proceed to the next phase, as this will introduce unacceptable risk to the overall program. Depending on the case, this phase should not last more than one to a few years maximum. The guiding environment should be maximally patterned after best available research studies, e.g. a federal research laboratory, a research university, an FFRDC, DARPA, etc.

4. Development contracts: Once all research teams achieve success (TRL of 3), issue the next phase RFPs in parallel to seek proposals from small expert development teams who can demonstrate past success and current readiness to perform the development of each needed module. Typically, the different modules will use completely different teams as the modules have little in common (the four boxes denoted "development" in Fig. 1 A). Since these teams will not have any unknown unknowns, these contracts must be fixed price, and the price should be guided by best commercial programs, with some reasonable overhead for handling military security and external management.

5. Design phase: when all modules have been developed, verified and validated, and are totally ready for system integration, issue an RFP for a larger contract to perform system design and integration, (denoted as "Design" in Fig. 1 A). This program will need engineers from all domain subsystems, as well as compe-

tent system-level engineers representing all relevant branches of engineering. This single contract should have a reasonably short schedule and fixed budget because all modules have been already created. (This is like a car design to use available engines, gearboxes, seats, radios, etc.) This phase should perform formal system-level systems engineering and program management, including integration, verification and validation. This phase is essentially a routine engineering system level design even for a new weapon or space systems, and must be treated as such, rather than as a bloated multi-year full R&D program. There should be practically no development but plenty of best design activities. Contractually, passing the buck between the different parties involved in phases 1-5 must be avoided, demanding that a green light into the next phase is contingent upon the acceptance of the previous phase. Coordination and communication opportunities throughout the program stakeholders and life cycle should be maximized.

6. Keep the contractor in phase (5) and the program manager fully accountable for the entire program technical and business success.

The above approach offers the following significant advantages:

a. Each project in each phase is manned in an optimized way, assigning only the experts and managers needed. We eliminate the "standing army" of thousands of highly paid engineers and managers for many years of "looking busy" while only a few individuals are truly needed. The cost of issuing one massive contract that mixes research, development and design is symbolically illustrated by the shaded area in Fig. 1 A. In contrast, separated and optimized research, development and design are like the small shaded areas in Fig. 1 B. Clearly the cost and time of the latter are significantly smaller than the former.

b. The folks best suited for each phase are used: systems engineers and architects for the concept phase, scholars for the research phase, developmental engineers for the development of modules, and design engineers for the remaining low-risk design phase. We eliminate the present practice of asking engineers to address scholarly challenges for which they are poorly suited and which they attack by massive and costly iterations.

c. Lower risk: the program split into these phases automatically assures healthy milestones. If even one phase fails to deliver, the program can be stopped and the phase re-bid with minimum waste in overall schedule and treasure.

d. The approach is much closer to the well-proven commercial practice, which costs one to two orders of magnitude less than the recent defense programs.

e. The shorter overall schedule is conducive to more stable requirements and the absence of technology changes during the program, the two aspects that have destroyed many a massive long defense program. Of course, the stability of customer-level need and use scenario requirements should be pursued by all means, as unstable requirements can destroy any long program.

4. The Mass Penalty

A careful reader no doubt noticed one technical deficiency of the above approach: namely that the modules predesigned for the design phase have to be used "as is", even if each is available in several size and shape combinations. The typical argument for contracting the entire program and all of its phases to a single contractor is based on the hypothesis that the contractor can then develop and optimize each module for minimum mass and best system layout. Theoretically there might be a merit in this argument. However, economics destroys it immediately, as follows: engineering labor rather than system weight is the most expensive item in large complex programs. Using pre-designed modules may carry a small weight penalty (which should be small indeed if the teams developing the modules understand the module use in the system of interest – not an unreasonable expectation), perhaps at worst requiring the system to be lifted into space by a slightly larger vehicle than what might be needed otherwise. For example, having to use a larger-size lift vehicle into space may cost an extra $50-$100 million dollars (a generous estimate), while the proposed approach will save billions if not tens of billions of dollars in much shorter program schedules. In addition, the proposed approach delivers the capability to the warfighter years ahead of traditional multi-year programs. It is simply common sense that this is a vastly better approach. Commercial programs understand it very well. Time for defense programs to do the same.

5. Summary

The proposed approach to complex weapon system development is based on clear separation of the program into research, then development, and finally design phases. Each phase should be performed using separate optimum-size teams of specialized experts, all coordinated by an efficient co-located small management team. The approach offers vast improvements over the current practice of one huge all-inclusive program lasting 10 to 20 years, costing a treasure, and wasting up to 90% of the cost or more because most engineers have really little to do most of the time, while a few are frantically trying to mature the TRL of selected modules using brute force iterations. Examples of poorly performing programs that started with low TRL have been cited. Examples have also been provided of successful programs that clearly separated research from development and from design.

The proposed approach has been practiced in the commercial world for tens of years. Thanks to it, we can buy a car for $20,000 rather than the billions it would cost to develop the car using the current defense contracting paradigm. The possible small added cost due to larger weight is compensated by orders of magnitude lower cost of engineering labor. The approach will yield higher affordability and faster availability to the warfighter. The approach is totally consistent with the Integrated Defense Acquisition Technology and Logistics Lifecycle Management Framework. Nothing in the present defense acquisition policy precludes the approach. Even the Program Objective Memorandum budget formulation [18] for defense programs which requires that military services perform program acquisition planning several years in advance could be adopted to handle the proposed program organization. A pilot program is recommended to follow the proposed approach. It has the potential to significantly cut the budget, schedule and bring the needed system into operations in a fraction of the current programs.

ABOUT THE AUTHOR

Bohdan "Bo" W. Oppenheim is a Professor of Systems Engineering at LMU. He is the founder and Co-Chair of the Lean Systems Engineering Working Group of INCOSE, co-leader of the effort developing Lean Enablers for Systems Engineering, author of Lean for Systems Engineering with Lean Enablers for Systems Engineering (Wiley, 2011) and the second author of the The Guide to Lean Enablers for Managing Engineering Programs (INCOSE, PMI, MIT LAI, 2012) . His engineering degrees include Ph.D., Southampton, U.K.; Naval Architect, MIT; MS, Stevens Institute of Technology; and B.S. (equiv.) from Warsaw University of Technology in Aeronautics. His credits include five books, 20 journal publications, $2.5 million in externally funded grants, and a 30-year industrial and consulting experience spanning naval, space, software and mechanical engineering. He is the recipient of 2011 Shingo Award, 2013 Shingo Award, 2010 INCOSE Best Product Award, 2011 Fulbright Award, and 2008 LACES Best Teacher Award.

Office: 310-338-2825
Home: 310-450-5713
E-mail: bohdan.oppenheim@lmu.edu

REFERENCES

1. M. J. Morgan, and J. K. Liker, Toyota Product Development System, Productivity Press, 2006.
2. GAO, Assessments of Selected Weapon Programs, GAO - 08 - 467SP, 2008
3. A. B. Carter, The Under Secretary of Defense, Acquisition, Technology and Logistics, Memorandum for Acquisition Professionals, June 28, 2010.
4. T. Hall, NPOESS Lessons Evaluation, ATR-2011(5558)-1, The Aerospace Corporation, El Segundo, CA, December 2010
5. Integrated Defense Acquisition Technology and Logistics Lifecycle Management Framework, DAU, <http://spacese.spacegrant.org/uploads/Project%20Life%20Cycle/DAU_wallChart.pdf>, accessed 09-25-2012
6. T. Rockwell, The Rickover Effect: The Inside Story of How Adm. Hyman Rickover Built the Nuclear Navy, John Wiley & Sons; 1995
7. S.B. Johnson, The Secret of Apollo, Systems Management in American and European Space Programs, John Hopkins, New Series in NASA History, 2002.
8. R. Leopold, The Iridium Story: An Engineer's Eclectic Journey, Minta Martin Lecture, MIT Department of Aeronautics and Astronautics, Apr. 23, 2004.
9. J. Oehmen, Ed., The Guide to Lean Enablers for Managing Engineering Programs, PMI-INCOSE-MIT LAI, 2012
10. B.W. Oppenheim, Lean for Systems Engineering with Lean Enablers for Systems Engineering, John Wiley & Sons, 2011
11. INCOSE LSE WG, Lean Systems Engineering Working Group website, 2012, <http://www.incose.org/practice/techactivities/wg/leansewg>.
12. D. K. Sobek II, A. C. Ward, and J. K. Liker, Toyota's Principles of Set - Based Concurrent Engineering, Sloan Management Review, Vol. 40, No. 2, Winter, 1999; pp. 67 - 83.
13. E. M. Murman, Lean Aerospace Engineering, Littlewood Lecture AIAA - 2008 - 4, Jan. 2008.
14. J. Warmkessel, Lean Engineering, Lean Aerospace Initiative, MIT, <http://lean.mit.edu>, 2002.
15. R. Rhodes, The Making of the Atomic Bomb, Simon & Schuster , 1986
16. MBSE, INCOSE, 2012, <http://mbse.gfse.de/>
17. VDM, 2012, <http://en.wikipedia.org/wiki/Vienna_Development_Method>
18. POM: <https://dap.dau.mil/acquipedia/Pages/ArticleDetails.aspx?aid=79420a26-7a89-4e94-aad2-6d5d61bb7511>, last accessed Oct. 22, 2012.

NOTES

1. For a description of TRLs see <http://esto.nasa.gov/files/TRL_definitions.pdf>, last accessed 9-23-2012.
2. Dividing this effort to more than one company (not unusual in mindless contracting focused on "spreading the wealth") is just as effective as cutting a person's brain into pieces, distributing the pieces and asking them to coordinate together- a suicidal proposition for program efficiency. The earlier GPS program suffered from it, burning budget and schedule on Interface Control Documents written by the 45 or so disjointed teams.

Efficiencies of Virtualization in Test and Evaluation

Elfriede Dustin, IDT
Tim Schauer, IDT

Abstract. Using automated testing in a virtual test environment can reduce the time and effort required to complete test execution and data analysis, significantly reduce test suite costs, and at the same time increase the thoroughness of system testing.

Section 1: Introduction

NIST produced a report in 2002 titled, "The Economic Impacts of Inadequate Infrastructure for Software Testing."[1] This report "estimates the economic costs of faulty software in the U.S. to range in the tens of billions of dollars per year and have been estimated to represent approximately just less than 1% of the nation's gross domestic product." The report goes on to state that "based on the software developer and user surveys, the national annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion."

Also in 2004 the Chief of Naval Operations (CNO) Guidance included direction to the Commander, Operational Test and Evaluation Force (COMOPTEVFOR) to lead a collaborative effort among Navy, OSD, and contractors to reduce the costs of Test & Evaluation (T&E) by 20%. In developing a response to the CNO Guidance for 2004, COMOPTEVFOR surveyed programs and included the following as T&E cost drivers:

- **Redundant testing**
- **Significantly increased levels of regression testing driven by technology insertion**
- **Increasing complexity of computer software testing, to include systems of systems**
- **Interoperability testing and certification**

Based on the COMOPTEVFOR findings, more effective approaches for testing are needed to be able to meet the CNO Guidance to reduce T&E by 20%.

A GAO Report to the Congressional Committees dated June 2012, describes that "recent defense acquisitions have experienced from 30% to 100% growth in software code over time."[2]

With the increased size and complexity of systems of systems testing, requirements for unique / duplicate test facilities and test-beds for major Navy product areas, software testing is rapidly becoming the "very longest and most expensive pole in the tent" when it comes to fielding new capabilities. Because of many reasons including organizational boundaries, lagging technologies, unique requirements, and testing methodologies, the testing of new capabilities being fielded has become a significant cost and time element of the process and without some form of change to the current process, could become even more significant. Some estimates have it consuming more than 60% of the time and cost of the process.[3]

Our experience at IDT shows that using virtual test environments with automated testing using Automated Test and Re-Test (ATRT) can help reduce testing infrastructure cost for testing areas such as interoperability, system testing, functional testing, component and unit testing. Additional benefits of automated testing in a virtualized environment include a more reliable system, improved testing quality, and reduced test effort and schedule. A more reliable system results from improved performance testing, improved load/stress testing, and improved system development life cycle through automated testing. The quality of the test effort is improved through better regression testing, build verification testing, multi-platform compatibility tests, and easier ability to reproduce software problems. Test procedure development, test execution, test result analysis, documentation and status of problems are also activities benefiting from automated testing.

ATRT in a virtual test environment can provide a stable, scalable, affordable and accessible automated testing infrastructure that extends across one or many server farms, across one or many System(s) Under Test (SUTs) and works with a common set of cloud computing concepts to support a broad virtualized enterprise automated test environment. This specific testing setup allows the use of virtualization in a specialized way to reduce the need for purchasing, storing and maintaining various expensive test environment hardware and software. Proper virtualization setup provides a multi-user access automated testing solution that allows users to implement and reuse ATRT, along with all testing artifacts, on a provisioning basis. Additionally all related automated testing activities and processes, i.e. test case and requirements import; requirements traceability, automated test creation and execution, and defect tracking take place in this virtualized environment.

Combining ATRT test efficiency with the hardware cost savings implementing in a virtualized/cloud environment, the resulting estimated savings are tremendous. For example 20 Virtual Machines (VMs) fit on 1 server in our virtual environment example – allowing for huge savings in the test environment, i.e. in this case a 20 to 1 cost savings. Additionally, in the virtual environment, the SUT VM can be located anywhere on a connected network and does not need to be located physically in the same VM as the testing VM.

Examples of automated software testing in a virtualized test environment include:

1. **Automatic provisioning of a virtualized automated test environment**
2. **Automatic provisioning of the entire automated testing lifecycle for any type of SUTs**
3. **Continuous integration using virtualized environments**

Sections 2.0 through 4.0 provide technical overviews of the various embodiments of the present ATRT/Virtual Test Environment (VTE) implementation.

Section 2: Automatic Provisioning of a Virtualized Automated Test Environment

As shown in Figure 1, the virtualized setup allows for a stable, scalable automated testing infrastructure that extends to one or many SUTs or one to many automated testing tool installations (in this example ATRT). This virtualized test environment setup is a highly scalable solution whether a user needs to run 10 or 10000s of tests connecting to N number of SUT displays and servers over days or weeks and whether the user needs to analyze 100s of test outcomes or 10000s or more.

In order to support a virtualized automated test environment, it is critical the automated testing solution itself be scalable. For example, the ATRT technology allows for N number of concurrent tests to run or N number of serial tests, depending on the test type required. All of the tests and test outcomes are stored in the ATRT database/repository for access by any subscriber (or user) of the ATRT virtualized environment. A subscriber/user can be a developer or tester or anyone on the program with ATRT user access privileges.

This virtualized test environment example setup supports live migration of machines; load balancing; easy movement of machines to different servers without network interruption and allows any upgraded VM to run on any server. As a result, it is also important that in a virtualized environment an automated testing solution is not only scalable but portable. ATRT can test systems independent of OS or platform so it is able to support applications running on both Windows and Linux providing flexibility to migrate machines without the constraint of the OS the automated test solution can support.

Additionally, an automated testing tool should be selected that does not need to be installed on the SUT. ATRT is an example of a solution that does not need to be installed on the SUT and instead is communicating with the SUT via a VNC Server or the RDP protocol which transmits the SUT images back to the tester to the ATRT client. Few tools exist that do not need to be installed on the SUT. The typical automated testing tool needs to be installed on the SUT so it can link to the GUI coding libraries to get the object properties of the GUI widgets and/or pull information out of the Operating System's window manager in order to create an automated test baseline. Installing an automated testing tool on the SUT however is generally not desired, because 1) the installation modifies the system environment (the testing system environment should be identical to the production system environment) and 2) it does not lend itself to cloud computing because of the additional tool installation on each SUT.

In this VTE the SUT VM can be located anywhere on a connected network and does not need to be located physically in the same VM as the ATRT VM. This allows for tremendous flexibility, for example multiple ATRT VMs can run in the VTE connecting to 100s of SUT VMs. However, in the typical automated testing setup where the tool needs to be installed on the same machine as the SUT, a 1 : 1 setup is required, i.e. 1 Automated Testing tool for each 1 SUT, negating some of the savings expected in a VTE.

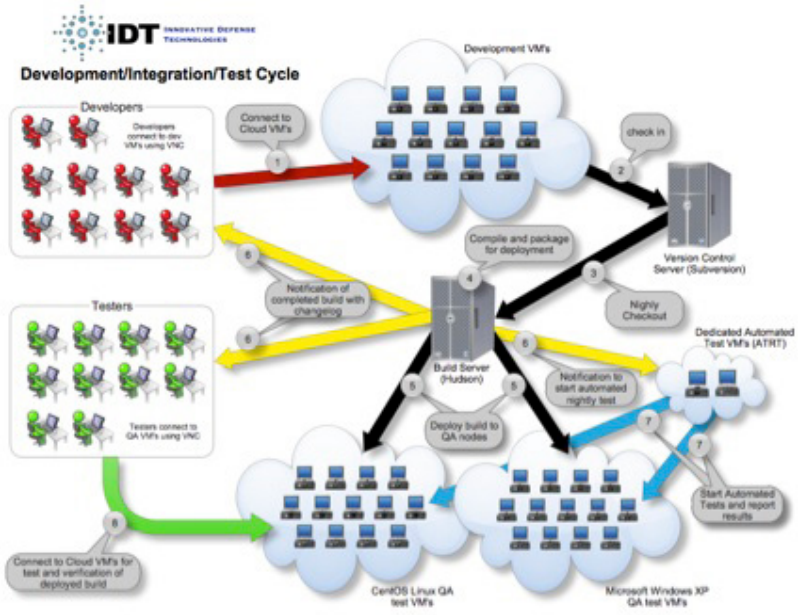


Figure # 1: Top-Level Block Diagram of the Automated Provisioning of the ATRT virtualized Test Environment

Section 3: Automatic Provisioning of the Entire Automated Testing Lifecycle for any Type of SUTs

One or many users can access a VTE one at a time or concurrently with any device such as a laptop, iPad, iPhone, etc. with nothing installed on their device but a network connection enabling the capability to login to an IP address to connect to the ATRT virtual environment.

Users can then request one or more instances of a VM along with the automated testing tool. The automated provisioning meets a user's changing needs without the users being required to make any software modification on their end as required to conduct the automated test. The VTE in this example can spawn an instance of ATRT which then allows the user to access any automated testing artifact and execute the automated testing lifecycle. The user can then conduct any activity that is part of the automated testing lifecycle, i.e. create an automated test case, reuse or troubleshoot an existing automated test case created by any user, import requirements, produce a requirements traceability report. The VTE provides any additional features and capabilities required to support the SQA process and help improve Quality, such as Unit Testing and Code Coverage.

Exemplary features of this process include:

- **Developers update the code on the development VMs**
- **Developers check in their code into Version Control**
- **Build Server conducts automated nightly checkouts**
- **Build Server compiles and packages a new build for deployment**
- **Nightly automated tests are run**
- **Users are notified of the automated test outcome**
- **Build Server deploys the new build to the QA nodes**
- **Testers access the QA nodes and create and/or run their automated tests**
- **Testers, Developers, and all users conduct the automated testing lifecycle activities and maintain all ATRT test artifacts in the virtual environment**

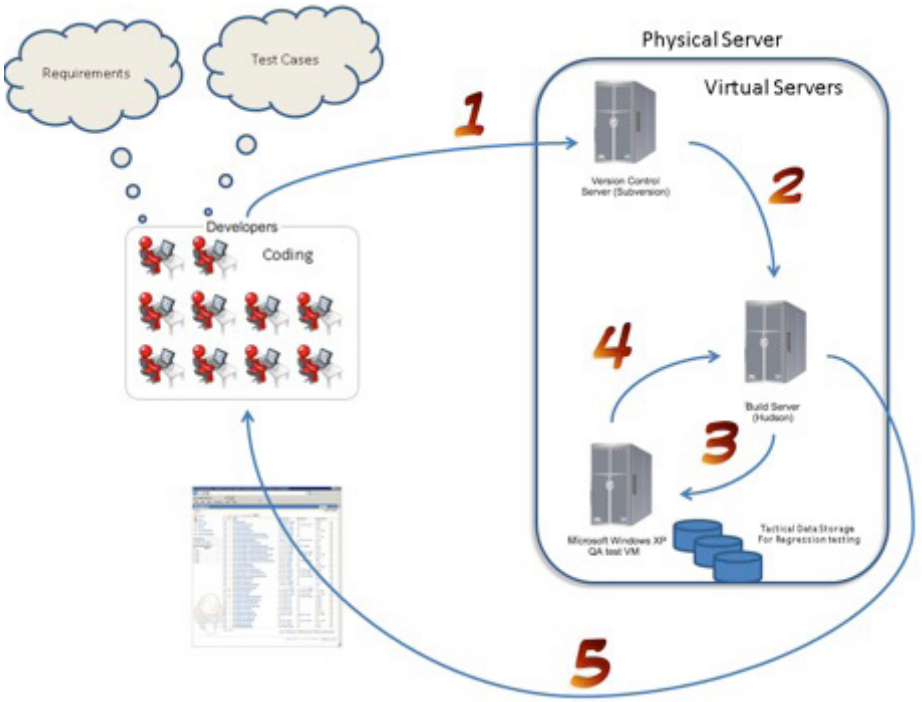


Figure 2. Continuous Integration Environment Example

Using the virtualized test environment a single engineer may control an entire test of complex systems with only his/her iPad, laptop, etc. and only requires access to the network.

Section 4: Continuous Integration Using Virtualized Environments

Continuous integration is an industry adapted software engineering best practice in which any change to the code or environment is tested and reported on as soon as feasible. In most cases this involves nightly software builds and nightly automated test runs to allow for quick look reporting on any newly introduced issues. Virtualized test environments play a major role in this best practice.

The development environment that makes this possible is one of a virtualized environment combined with both regular workstations and laptop computers networked together.

1. Developers first review the system level requirements and create a set of automated tests. Code is locally edited / compiled/linked and then checked in to a virtualized version control repository, such as SVN. From here other developers can check out both updated code off of the trunk or from code from specific branches to support different build.

2. Upon code checkin, a continuous build server, such as the Hudson Continuous Build virtual server is triggered to start a complete build/check/test/report cycle. Hudson will perform the following tasks:

a. Update the latest code from SVN

b. Compile the code and check for compile errors

c. Link the code, check for any link errors

d. Perform source code style checks and copyright checks

e. Start a series of both internal and external regression tests:

i. Internal regression tests will execute automated tests to verify key use case tests to verify results are as expected and also ensure that code that was updated has not adversely affected the existing functionality.

ii. External regression testing can then utilize any automated testing capability on another virtualized node to perform tests as an end user would be expected to do (i.e. through a GUI interface). Each test can then analyze hundreds of system level requirements. Each requirement may itself be verified hundreds to thousands of times. External regression testing again compares its results against a known good set of results.

3. The internal and external testing results are then reported back to the Hudson server. Upon completion of successful internal and external regression testing, the Hudson server continues to now build an installer package that will be available to the end user at fielded locations. Additionally, key statistics are gathered on the entire process and saved for later retrieval.

4. Finally, Hudson provides the developer with reports on the entire sequence of testing. The developer can then use the results of the testing to make appropriate code changes.

Section 5: Summary

Using automated testing in a virtual test environment we have been able to demonstrate the ability to reduce the time and effort required to complete test execution and data analysis, significantly reduce test suite costs, and at the same time increase the thoroughness of system testing. An increase in software testing thoroughness equates to a reduction of defects found in the field and reduced total ownership cost. Automated testing in a virtualized test environment will also enable much earlier identification of integration and interoperability characteristics of any software products that must interact with other systems. Identification of software specific integration characteristics in products in-stride with software development cycles enables the identification of issues to also be decoupled from the delivery of the final hardware configuration.

NOTES

1. See <http://www.nist.gov/director/planning/upload/report02-3.pdf>
2. <http://gao.gov/assets/600/591608.pdf>
3. Hailpern and Santhanam, 2002 (The cost of providing [the assurance that a software program will perform satisfactorily in terms of its functional and nonfunctional specifications within the expected deployment environments] via appropriate debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development).

ABOUT THE AUTHORS

Elfriede Dustin is Director of Solutions at IDT where she works on developing new ideas and discovering new approaches to the requirements based automated software testing challenge. Software development is still an art and that makes automated software testing a special challenge. IDT (www.idtus.com) strives to meet that challenge by producing a reusable automated testing framework that includes reusable automated testing components, starting with requirements through the entire software testing lifecycle to defect closure. Elfriede has a B.S. in Computer Science with over 20 years of IT experience, implementing effective testing strategies, both on Government and commercial programs. She has implemented automated testing methodologies and testing strategies as an Internal SQA Consultant at Symantec, worked as an Asst. Director for Integrated Testing at the IRS Modernization Efforts, implemented testing strategies and built test teams as a QA Director for BNA Software, and was the QA Manager for the Coast Guard MOISE program.

She is the author and co-author of 6 books related to Software Testing, i.e. author of the book "Effective Software Testing" and lead author of "Automated Software Testing" and "Quality Web Systems," and co-authored the book "The Art of Software Security Testing," together with Chris Wysopal, Lucas Nelson, Dino D'ai Zovi, which was published by Symantec Press, Nov 2006.

Together with IDT CEO Bernie Gauf and IDT FSO and Sys Admin Guru Thom Garrett she wrote her latest book "Implementing Automated Software Testing."

E-mail: edustin@idtus.com

Tim Schauer graduated from the University of Wisconsin-Madison in 1985 with a Bacholor of Science degree in Physics and a B.S. in Astro-physics. He received his commission in the US Navy and worked as both the weapons officer and communications officer on the USS Shenandoah. After the Navy, Mr. Schauer worked on Tactical Software for the SPY-1A Phased Array radar at the Naval Surface Weapons Center in Dahlgren, VA. He then became testing lead and lab manager for the SeaWolf Class / BSY-2 integration facility in Moorestown, NJ. Later, he worked as senior logistics analyst for US Pacific Command at Camp Smith, Hawaii.

Tim Schauer has been working with Virtual Servers since first being introduced to them at Pacific Command (PACOM) in the late 1990's. He continued to develop virtual systems while working at the San Diego Data Center for the County of San Diego and Children's Hospital of Los Angeles. He has virtualized over 90% of the Beaufort County, South Carolina, library's IT system, greatly reducing cost while increasing productivity. Finally, Tim is currently working on virtualizing a US Navy Tactical Weapons System to facilitate ongoing ATRT automated testing at the IDT facilities in Arlington, VA.

E-mail: tschauer@idtus.com
Phone: 843-473-5465

Upcoming Events

Visit <http://www.crosstalkonline.org/events> for an up-to-date list of events.

GFIRST 2013
25-30 August 2013
Grapevine, TX
http://www.us-cert.gov/GFIRST

AUTOTESTCON 2013
16-19 August 2013
Schaumburg, IL
http://www.autotestcon.com

APCOSEC 2013
9-11 September 2013
Yokohama, Japan
http://www.incose.org/newsevents/events/details.aspx?id=190

Defense Systems Acquisition Management Course
16-20 September 2013
Kansas City, MO
http://www.ndia.org/meetings/302E/Pages/default.aspx

Software and Supply Chain Assurance Forum
17-19 September 2013
McLean, VA
https://buildsecurityin.us-cert.gov/swa

(ISC)2 Security Congress 2013
24-27 September 2013
Chicago, IL
https://www.isc2.org/congress2013/default.aspx

World Congress on Engineering and Computer Science
23-25 October 2013
San Francisco, CA
http://www.conferencealerts.com/show-event?id=112271

16th Annual Systems Engineering Conference
28-31 October 2013
Arlington, VA
http://www.ndia.org/meetings/4870/Pages/default.aspx

OWASP AppSec USA 2013
18-21 November 2013
New York, NY
http://www.sourcesecurity.com/events/free-event-listing/owasp-appsec-usa-2013.html

SUBSCRIBE TODAY!

To subscribe to CrossTalk, visit www.crosstalkonline.org and click on the subscribe button.

Twenty-five Years of the Wrong Choices!

Just a few weeks ago, I was notified that my article submission for the 25th Anniversary Edition of CrossTalk was accepted—so somewhere in this issue, there is my scholarly article about how we have progressed over the last 25 years. However, this BackTalk column is a fitting end to the issue. It is more about where we have digressed since 1988. After all, in retrospect, it is easy to see where we made the right choices. It is where we made the wrong choices that few people wish to elaborate on.

In 1988, I was the proud owner of a really high-quality video tape recorder, chocked full of awesome features, complete with stereo recording. It had a digital channel selector, and could program up to 12 (that is right – TWELVE!) future recordings. A high-quality electronics manufacturer, Sony, made it. And it was a Betamax. Arguably, a better product than VHS—it had stereo and higher quality video—but it was a losing battle.

Of course, I had a "backup" format for the movies that I found really important—ones that I paid money to buy, so that I could have a high-quality movie that I could watch over and over, for years and years to come. Yes, I owned a LaserDisc. I had LaserDiscs of "The Wall" and "Rocky Horror Picture Show."

Back in 1991, I graduated from Texas A&M with my Ph.D. I took not one but two courses on parallel algorithms and parallel sorting. It was not a question of, "if we would be converted to parallel processing by 2010." It was more of a question of, "what kind of parallel architecture would we all be using?" Choices included the mesh, the cube, and the butterfly, just to name a few. Granted, we now use multi-core, multi-threaded machines, but few programmers really know how to write code to truly take advantage of parallelism. Instead of large-scale parallelism, we now do parallelism "in the small"—nothing at all like what we envisioned back in the early 1990s.

Also in the early 1990s we thought that by 2000, there would really be only one programming language used in the DoD, right? Heck, I was a member of the Ada Government Advisory Group (a.k.a. the Ada GAG—a horrible acronym if there ever was one).

In the mid 1990s, I was convinced that the 3.5" floppy disk was eventually going to disappear – the thin floppy was incapable of holding enough information – so I made sure to back up everything I had on the one medium that we just knew would be around for years and years to come—the Iomega Zip Drive.

By the year 2000 came along we decided that the "single programming language" idea was never going to work, so we decided to agree on a common operating system instead. I was on the working committee for the Defense Information Infrastructure Common Operating Environment (DIICOE). Bet you have not heard of DIICOE in a while either, have you?

Even though I am the epitome of a die-hard Mac user, for about 15 years, I used another OS. What did I switch to? Linux, of course. In the 1990s, we just knew that by the early 2000s, Linux would be the predominant operating system for both home and office.

Speaking of the Apple Macintosh, who would have predicted that both Macs and PCs would share the same chips? Over the years, I learned and then taught 6800/68000 assembly language, and also mastered the Power PC (PPC) architecture. Now, my Mac runs on an Intel, and using a virtual machine interface, it boots either OS X, Windows 7 or Windows 8.

I only represent one lowly software engineer—and the list of projects, technologies and initiatives I have been on that are obsolete and no longer part of the DoD is really long. One could argue spectacularly long. So, this means I have been a failure, right?

Well … no, to put it bluntly. In fact, almost everything I have listed above actually contributed to progress in engineering and computing science. Ada is still used, and some of the features it heralded became part of other, newer languages. Parallel processing is still a critical component of supercomputing. In fact, it appears that Moore's Law might apply to the number of processors in a system. DIICOE helped us standardize some critical components of embedded operating systems, and helped standardize some real-time operating system. The Un*x OS is not extremely popular for home computing, yet it runs a lot of servers, supercomputers, and large-scale systems. It is also the basis for the Mac OSX operating system.

What about the 68000 and PPC architectures? They are used in high-speed embedded systems. The LaserDisc? The DVD simply eclipsed it—higher capacity, smaller size, cheaper technology, and better quality video. Same with the Zip drive. It was great for its brief time, but the non-moving technology (and eventual greater capacity) of the USB drive sounded its death knell. These were not failures, just technologies that were eclipsed by better technology. There is no shame in having worked on a once cutting-edge technology that becomes obsolete.

That is just the way progress is. Two steps forward, one step back. Every great new technology we have today is based on something that preceded it. You cannot judge progress by the number of technologies that have failed and been replaced. You can only say "What we have now is better than what we had yesterday."

Learn, improve, discard, and move on. I would bet that every decent developer or software engineer could point (usually with pride) to some project they worked on that has been made obsolete by the steamroller of progress. And every one of us has learned from the experience.

Progress marches on.
Just like CrossTalk.
Happy 25th Anniversary!

David A. Cook
Stephen F. Austin State University
cookda@sfasu.edu

CrossTalk / 517 SMXS MXDEB
6022 Fir Ave.
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

CrossTalk thanks the above organizations for providing their support.